Cerebral Data Structures:
Integrating Context into Data Structure Design and Implementation

a dissertation presented

by

Brian Hentschel

to

The School of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

April 2022

Dissertation advisor: Stratos Idreos Brian Hentschel

Cerebral Data Structures:
Integrating Context into Data Structure Design and Implementation

Abstract

The speed at which computer programs execute operations is fundamental to the way we build applications, with faster performance lowering operational costs and creating better user experiences. In addition, faster performance directly translates to the feasibility of new applications for complex problems. A key component of the overall performance of computer programs is the performance of their data structures, the structures which control how data is stored, accessed, and modified by the program.

The traditional paradigm for creating data structures is to design them without formal knowledge of their context. This includes two modes of design. In the first, data structure are designed for broad applicability through worst-case guarantees which hold over arbitrary input data. In the second, more complex data structures are specialized to the system at hand, with the designer's intuition implicitly guiding the design of the data structure towards one they believe performs well on their context.

The goal of this thesis is to move past the prior paradigm towards a paradigm where learning about context, i.e. data, query, and hardware properties, is explicitly integrated into the design process of data structures. This is done by 1) choosing context properties to estimate from samples of past data and operations, and 2) using these estimated properties to control the shape and algorithms of data structures. The results show the possibility for fundamental improvements, with data structure designs that depend on data showing theoretical and experimental performance improvements on a large class of workloads when compared to designs which lack knowledge of context. Specifically, we demonstrate 10-100X improvements in filter data structures, scans over arrays, and hashing structures when compared to state-of-the-art designs from research and industry.

Contents

Acknowledgments

This thesis would not have been possible without the guidance of my advisor, Stratos Idreos. I am grateful to him for spending many hours throughout my Ph.D. mentoring me. From him, I learned much about how to craft research projects, how to communicate research ideas effectively, and how to be a collaborator and mentor to others. I also learned the many ways I was presenting slides incorrectly, and slowly over time learned to make slides beautiful enough to make him happy. I have been extremely fortunate to have an advisor so willing to spend his own time to make me a better researcher.

I am grateful additionally to the many mentors I have had outside of Harvard in my growth as a researcher. A special thank you to Yuanyuan Tian and Peter Haas, who mentored me during a summer internship in my first year at Harvard and who have remained as steadfast collaborators of mine throughout my Ph.D. It is with them I experienced my first paper deadline (a thrilling experience!). While our work together is not part of this thesis, I consider it an integral part of my PhD.

I am thankful as well for the many members of DasLab who have been with me throughout my years: Abdul Wasay, Michael Kester, Lukas Maas, Kostas Zoumpatianos, Wilson Qin, Manos Athanassoulis, Niv Dayan, Subarna Chatterjee, Utku Sirin, Kyle Deeds, and Sanket Purandare. The camaraderie I felt with my lab has been a source of infinite fun and support throughout the years, and made my studies at Harvard all the more worthwhile. Additionally, this relationship led to a lot of great chicken being eaten.

A big thank you as well to all of my friends outside the lab at Harvard. In no particular order, I want to say thank you to Bernd Huber, Dimitris Kalimeris, Keye Tersmette, Sebastian Gehrmann, Samantha McBridge, Anastasiya Belyaeva, Tanvi Ranjan, Yamini Bansal, Michael Han, Brant Hoffman, Casey Meehan, Solsire Cusicanqui, Maleah Fakete, Verena Volf, and Baptiste Lemaire. My time at Harvard would not be the same without all of you.

Thank you to my partner Sharon Qian for being a constant source of support and fun throughout my time at Harvard. I am very grateful our walks around Harvard Yard to take breaks from work turned into much more. I look forward to many more years together.

Finally, I want to thank my family for providing support for me throughout my entire life. I am grateful to my sister, Lauren, for being a goofball with whom I am happy to have had to share my entire life with. I believe you taught me through excessive goofiness that there are no stupid ideas. And lastly, I am grateful to my parents, Naomi and Craig, for all the hours they spent feeding me, bathing me, and taking care of my every whim as a child so that I could focus on learning. It is from them I learned to love learning, and this degree is a reflection of all the effort they spent on furthering my education.

Chapter 1

Introduction

Data systems and structures are crucial in the process of turning the world's information into insight. They provide the infrastructure which allows analysts to ask questions through easy to use interfaces and expect reasonable response times on reasonable budgets.

A key problem facing data systems is that each year we want more computation from them. For instance, it is well documented that systems need to process an exponentially growing amount of data each year [Reinsel et al., 2018]. Beyond the growth of data, systems also need to process a growing number of queries [Djuraskovic, 2019, Sullivan, 2016]. Additionally, the analyses we want to do are more complex, with the advent of more complex but more accurate or optimal algorithms from machine learning and optimization.

The current solution to address the growing need for computation is to depend on advances in computer hardware and on the engineering support to build efficient data systems for this new hardware and the newest types of analyses. This approach has worked well enough to support a burgeoning information technology (IT) sector and allow for the service of many new applications, but it has its limits. In recent years developments in hardware plus engineering efforts have not been enough to curb overall IT spending: each year organizations across both industry and science need to spend more running their software stack [Rimol, 2022, Choi, 2020]. Additionally, the overall performance of data systems is limiting progress in certain applications: for instance, in sciences such as genomics and astronomy, many of the limits to what can be discovered now lie in the computational expense of querying large datasets to discover phenomena [Zhang and Zhao, 2015, NIH, Stephens et al., 2015].

An exciting vision towards solving this problem of the increasing cost of software systems is to create instance optimized systems. That is, instead of building systems which behave nearly the same for each application scenario, design systems which optimize themselves to the context at hand (i.e. things such as the incoming data and queries as well as provided hardware). This vision has been revisited numerous times over the past 3-4 decades and in many forms. For example, early work on component-based systems proposed building systems from plug and play components and then adapting systems by using or not using particular data structures as part of systems design [Batory and O'Malley, 1992]. More recently, there has been work on customizing base data layouts to context: for instance, in relational systems by adapting between data layouts such as row-store, column-stores and hybrids [Alagiannis et al., 2014, Arulraj et al., 2016]. Additional work has looked into creating systems that automates physical index tuning in a way that is reactive to the workload [Idreos, 2010]. Over the last couple of years, research efforts are now focusing on the larger vision: how to fully customize data systems across all critical parts of their designs by using combinations of learning techniques [Kraska et al., 2019] and parameterization of their possible design spaces [Chatterjee et al., 2021, Idreos et al., 2019].

Data systems are complex pieces of software with multi-million line code bases. There are numerous components involved and instance-optimized systems need to adapt many of those components. For example, recent work is looking into enriching through machine learning diverse components such as query optimizers [Marcus et al., 2021], caching [Liu et al., 2020, Chłędowski et al., 2021], and physical index selection [Ding et al., 2019]. In addition, recent work also looks more holistically into the whole problem of tuning a complex data system and all its knobs using machine learning resulting in design that are often superior than those of humans [Van Aken et al., 2017].

Data structures are one of the most crucial components. Because all computer programs write, access, and modify data as part of computation, the performance of their data structures is an integral part of the performance of the overall programs. For instance, hash tables in just C++ are

2% of overall CPU cost and 5% of overall memory cost at Google [Kulukundis], Bloom filters take up a large portion of the memory allocation for LSM trees [Chatterjee et al., 2021], a majority of the time spent in analytical SQL queries is on hash table joins [Sirin and Ailamaki, 2020, Dreseler et al., 2020], index traversal plays a key role in the cost of OLTP transactions [Zhang et al., 2016], and sketches are a computational bottleneck in network switches [Liu et al., 2019].

Our work in this thesis focuses on creating instance optimized data structures. Given a context, our goal is to create optimal data structures that maximize performance and maintain good systems properties such as robustness. We start by reviewing existing approaches.

1.1 Existing Approaches in Data Structure Design

The performance of data structures, such as their memory requirements or the latency of their operations, is a function of both the data structure implementation and of the individual operations given to the data structure. While the operations are generally viewed as unchangeable, the goal of engineers is to design data structures which efficiently support the target workload. We cover three approaches towards designing these data structures, which we think of as "the traditional approach", learned data structures, and adaptive data structures.

1.1.1 The Traditional Approach

The most common approach to fitting data structures to workload is to use a combination of engineer intuition and to combine this with rigorous analysis of the worst-case performance of the data structure over arbitrary data distributions and operations. For instance, for canonical implementations of data structures such as hash tables, we have big-O guarantees on the performance of each operation conducted by the data structure (which hold regardless of data inputs). Then, for constant factor optimizations, engineers make micro-level decisions about whether to prioritize operations such as inserts or reads, and whether to prioritize metrics like operation latencies or memory requirements. Even when building more complex data structures targeted for specific workloads, this approach of programmer intuition and worst-case bounds is still the common approach. Giving a specific example, MassTree uses a combination of tries and B-trees to support point and range gets as well as inserts efficiently for a target workload of long URL prefixes, and then backs up the data structure with worst-case bounds over arbitrary workloads [Mao et al., 2012]. In general, this approach of using engineer intuition and worst-case bounds is extremely robust, making the produced data structures easy to use and deploy in applications.

However, the main drawback of this approach is that 1) each time we want to specialize a data structure to context, a human needs to use their intuition to do so, and 2) that by thinking in terms of arbitrary inputs we miss out on potential performance benefits by tailoring the data structure to the realistic inputs it is likely to see.

1.1.2 Learned Data Structures

Learned data structures consist of using machine learning models to mimic or to augment traditional data structures [Kraska et al., 2018, Ferragina and Vinciguerra, 2020, Ding et al., 2020]. This approach has produced significant benefits for certain operations, specifically when models which are computationally cheap can well-approximate desired functions of interest. For instance, numerous works have shown the ability for integer datasets to approximate quantile functions with piecewise linear models instead of through the traditional approach of storing elements of a target set [Kraska et al., 2018, Ferragina and Vinciguerra, 2020, Kipf et al., 2020, Crotty, 2021]. This allows for replacing internal nodes of data structures such as B-trees with functional models which require less space and can lead to faster traversal of internal nodes. Another example is from filter data structures, where classification models are used as a source of side information to create filters with better space/time tradeoffs [Kraska et al., 2018, Dai and Shrivastava, 2020, Mitzenmacher, 2018].

A weakness of learned data structure approaches is that more complex decision boundaries create large computational overheads. For instance, in the case of filters, computationally complex classifiers produce 184X slower query speeds compared to using traditional Bloom filters [Deeds et al., 2020]. Similarly, learned order-preserving indexes are less efficient on string datasets which have more complex cumulative distribution functions than integers [Spector et al., 2021]. Additionally, changes in the approximated function which can occur as a function of inserts to the dataset or as changes to queries can cause degradation in performance and so learned approaches tend to be less robust.

1.1.3 Adaptive Data Structures

Adaptive data structures are another approach for achieving context-specific data structures. The approach is organized around the idea of using operations needed by the data structure to change the way the data structure operates. For instance, database cracking makes use of queries to incrementally build index structures, building only the parts needed by the queries in question and saving on index construction costs [Idreos et al., 2007]. A second example is that of adaptive filters, where the filter structure in question changes its structure on false positives to reduce the probability of future false positives from the same query [Mitzenmacher et al., 2020, Lee et al., 2021, Bender et al., 2018]. While adaptive techniques vary in terms of which performance metric they benefit, they are tied together by the idea of using data structure operations to implicitly learn about the workload for the data structure and to optimize for that workload.

Adaptive structures give great benefits in that they do not need to learn a workload up-front, but they generally make decisions we would have made if we knew the workload in advance. Thus, they do not unlock fundamentally new designs in data structures but instead adapt data structures towards known designs that fit the current workload. For instance, database cracking brings in the idea of adaptively building only the parts of an index that are necessary (an obvious decision if we knew which parts of the data are necessary beforehand!), but generally adapts towards a known context-agnostic or context-specific data structure. Similarly, adaptive filters try to adapt the filter to lower the false positive probabilities of recently seen queries, an approach similar in nature to those of filters which learn about the workload beforehand (such as Weighted Bloom filters, Stacked Filters, and Learned filters) [Bruck et al., 2006, Deeds et al., 2020, Kraska et al., 2018]. As a result, we can think of adaptive data structures as a technique for bringing flexibility into when workload knowledge is gathered and as a technique to adapt to shifting workloads, but which requires beforehand a context-specific data structure design to adapt to.

Along with their main benefits of being able to adapt to changing data distributions and not needing knowledge beforehand, adaptive data structures have significant drawbacks as well. In particular, they have to pay the cost of adaptation, which can be significant, and need to perform all learning online, which is significantly harder as a problem generally than learning offline.

1.2 Cerebral Data Structures: A Parametric view of Context-Specific Data Structure Design

Context-specific designs should give the best performance by definition: their design is specifically tailored for the use case at hand. However, learned data structures computational expense and lack of robustness mean that they are often not better than "traditional" designs, whereas for adaptive data structures they are generally adapting between already known designs (context-specific or traditional) and need to pay the cost of adaptation. Thus, while both are tremendously useful in data structure design and have many use cases, they leave holes in that there are many data structures where no known context-specific design produces meaningful improvements over designs which work over arbitrary data. This raises several key questions: 1) can we understand when context-specialization will give us substantial benefits over traditional designs, and 2) what are the design options we control in terms of changing data structures to give us the best context-specific performance.

While both these questions are not possible to answer fully, the goal of this thesis is to make meaningful progress on both by showing a new approach to context-specific designs that outperforms traditional, learned, and adaptive designs for major classes of data structures. We call this approach Cerebral Data Structures. Towards introducing Cerebral Data Structures, we start with a high-level restatement of the overall goal of data structure design. As data structure designers, we have control over the data structure implementation but not the operations given to the data structure. At the same time, the performance of the data structure is a function of both. Thus, our goal is given the fixed context of workload operations, to choose a data structure design which is in some sense optimal for the context.

Our approach is to formalize this optimization problem, and through doing so, enable a new framework for data structure design. In our approach, the workload of operations is viewed as random, and we are interested in finding parameters of the random distribution of workload operations such that under our model, much of the dependence of the performance of our data structures on the workload is captured by the parameters. These parameters are then estimated from samples of past operations. In conjunction with the introduced parameters, our goal is to introduce new designs of data structures which take advantage of the value of the estimated workload parameters. We place no limitations on the designs of these data structures, and note that the designs can include components of both learned and adaptive data structures, but also that quite often creating new designs by combining elements of context-agnostic data structures in novel ways produces excellent performance. Using this formulation of the problem, choosing a context-specific design becomes the problem of finding which data structure design obtains the best performance given the parameters of the workload.

This approach moves forward the goals of designing context-specific data structures in two key ways. First, by being rigorous in our approach to understanding how parametric models affect data structure components, we can understand better the types of assumptions necessary to create significantly better performance in data structures. Second, we show that creating context-specific data structures does not need to require machine learning models inside the data structure. In particular, we show several designs where all learning happens from the estimation of the parameters of the workload, and that model-free data structures can be designed to take advantage of these parameters which often match many of the best parts of learned data structures without having their drawbacks.

1.2.1 Designing a Cerebral Data Structure Step by Step

To make the framework of Cerebral Data Structures more concrete, we introduce now a step-by-step breakdown of the methodology of Cerebral Data Structures. This methodology still requires human ingenuity, but the goal of the framework is that by adding structure to the creative process, we allow for the faster discovery of new designs.

Step 1: Identification of Important Metrics. The first step in Cerebral Data Structures is to identify which metrics are of importance for the given problem (memory, computational speed, accuracy, robustness,...), and to characterize what components of the data structure are responsible for those metrics. For instance, in a hash table, the metrics are speed, memory consumption, and robustness. The components which contribute to the overall performance are the hash function, including its computational speed and collision probabilities, the storage of keys and values (which influences memory and also speed via cache misses), the method for handling collisions, and the cost of comparing keys which do collide. Often, based off empirical profiling of a data structure, it is known that certain components are more influential in overall metrics than others.

Step 2: Context-Specific Data Structure Design. We divide step 2 into two intrinsically connected steps. Step 2a is to examine how workload assumptions, or lack thereof, influence the design of these important components, whereas step 2b is the goal of finding a realistic assumption that allows for a more performant component of the data structure. As an example, we examine how Cerebral Data Structures looks at the hash function in hash tables. The goal of the hash function is to provide uniform outputs in a computationally performant manner. Without knowledge of context, the design of hash functions has to provide this uniformity in output over arbitrary keys. As we show later in Section 3, by providing structure to keys by assuming certain bytes are random, we can provide uniform outputs while being significantly computationally cheaper, thus making overall more computationally efficient hash tables. One could also use the Cerebral framework to examine other components of hash tables such as collision resolution, how to redesign the comparison operation, or compression for hash tables, with an eye on how parametric assumptions about the workload would allow for optimizing each component. The eventual goal should be to show via concrete equations how the parametric assumptions about the workload lead to greater performance in some aspect of the data structure.

Step 3: Estimation of Parameter Values. The third step in Cerebral data structures is to create estimators for the required parameters of the workload. This generally means creating a function which takes in samples of past data and produces estimated values for the parameters required. Building on our running example, for hash function design this estimator of parameter values is a procedure which identifies which bytes are random and how random they are given samples of past data and queries.

Step 4: Instantiation. The fourth step in Cerebral Data Structure design is to go from estimated values for the parameters plus some runtime info to a concrete instantiation of the data structure. For instance, in our running example, this means taking our estimates of which bytes are random and how random they are as well as runtime info such as the size of data structure required and using this to produce the hash function for the data structure.

1.2.2 Benefits of Cerebral Data Structures

The approach of Cerebral Data Structures has two main benefits. First, by requiring a formal parametric model of the workload, data structure designs can argue formally about their expected performance on a workload. This enables easier comparison of data structures, and introduces an operational understanding for users of data structures of when the context-specific design will bring performance benefits. In particular, by formally stating workload assumptions, users of data structures can more easily understand whether their workload fits these specified assumptions and decide whether using the context-specific design will bring meaningful performance improvements.

Second, the approach allows very broadly for creativity in data structure design. It allows for adaptive and learned data structures, but also allows (and indeed encourages) about the rethinking of various components of classical designs in data structures. As we will see, this rethinking of classical designs allows for data structures which are fit to context but do not use models inside the data structure, with these designs maintaining many of the benefits of learned approaches to data structures while providing superior computational performance as well as to robustness. Thus, the framework of Cerebral Data Structures meaningfully broadens the set of possible data structure designs.

1.3 Overview and Contributions

The rest of this work focuses on building out the framework of Cerebral Data Structures through applications to several fundamental data structures. Through these examples, we give support to the main thesis of this work:

Thesis: Formalizing parameters of the workload and reasoning through how they affect data structure designs is crucial to designing performant and usable context-specific data structures.

This thesis then leads naturally into what we feel is our primary contribution, a framework for designing data structures which rely on explicit parameterized models of the workload. To support our primary contribution, we show that this framework leads to designs which produce significant improvements on three classes of fundamental data structures: hash functions with a focus on hash tables (Ch. 3), filter data structures (Ch. 4), and predicate evaluation over unsorted arrays, a central component of analytical databases (Ch. 5).

We choose these three structures to focus on because of their significant importance across software systems including data systems. Hash tables are by and large the default way to access data across all programs, and as a result improvements in their performance enable improvements across general computer programs. For filter data structures, for many systems with data that resides on disk or across network, the communication costs to get unneeded data is a major part of their overall costs. And for analytical data systems, the cost to access their base data and evaluate predicates on it is a baseline for the overall performance of the system as all analytical queries start by selecting data to work over.

Before delving into each of these projects in detail, we introduce related background (Ch. 2), and then afterwards, we talk about how to design new Cerebral Data Structures (Ch. 6). We then conclude the thesis by restating many of the goals of Cerebral Data Structures and reflecting on how the introduced ideas move forward these goals (Ch. 7).

1.3.1 Contributions

We believe the contributions of this thesis to be:

- *Cerebral Data Structures*: A framework for designing context-specific data structures that works by creating formal parameterizations of data structure workloads and reasoning through how these workload parameters affect data structure designs. (Ch. 1)
- *Entropy-Learned Hashing*: We apply the concept of Cerebral Data Structures to hash functions and show how estimating the entropy of specific byte locations of incoming data items leads

to hash functions which hash only parts of incoming keys, reducing hash computation, while preserving the uniformity of hash function outputs. (Ch. 3)

- *Hashing Performance Improvements*: We show that Entropy-Learned Hashing provides higher throughput than traditional hashing with improvements of up to 3.7× for hash tables, 4.0× for Bloom filters, and up to 14× for data partitioning on medium-sized key types such as URLs. We show improvements of up to 228X on large 8KB keys. (Ch. 3)
- *Stacked Filters*: We apply the concept of Cerebral Data Structures to filter structures, showing how identification of frequently queried items combines with stacking filter structures to create filters which have better false positive rate vs. memory tradeoffs than traditional filters while being robust to workload shift. (Ch. 4)
- *Filters Performance Improvements* We show experimentally that Stacked Filters provide improvements in FPR of up to 100× at the same memory budget over the best traditional filter designs, while they provide similar false positive rates to learned approaches in filtering while being up to 184X faster. (Ch. 4)
- *Column Sketches*: We apply the concept of Cerebral Data Structures to database scans, showing how creating an auxiliary index which represents items via a lossy approximation of the CDF improves the performance of predicate evaluation over unsorted arrays regardless of how many data items satisfy the predicate, the distribution of data values, and the clustering of data values. (Ch. 5)
- *Scan Performance Benefits and System Integration* We show that using Column Sketches is a robust way to improve scan performance across a range of queries, and is up to 4X faster than the best alternative approach. (Ch. 5)

1.3.2 Published Papers

The material in this thesis is based in large part on the following publications:

- Brian Hentschel, Utku Sirin, and Stratos Idreos. Entropy-Learned Hashing: Constant Time Hashing with Controllable Uniformity. *In Proceedings of the ACM SIGMOD International*

Conference on Management of Data, 2022, Philadelphia, PA, USA, June 12-17, 2022. [Hentschel et al., 2022]

- Kyle Deeds*, Brian Hentschel*, and Stratos Idreos. Stacked Filters: Learning to Filter by Structure. *In Proceedings of the VLDB Endowment 2021*, pages 600–612, December 2020. (* denotes equal contribution) [Deeds et al., 2020]
- Brian Hentschel, Michael S. Kester, and Stratos Idreos. Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation. *In Proceedings of the ACM SIGMOD International Conference on Management of Data, 2018*, Houston, TX, USA, June 10-15, 2018. [Hentschel et al., 2018]
- Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models, *In Proceedings of the ACM SIGMOD International Conference on Management of Data, 2018*, Houston, TX, USA, June 10-15, 2018. [Idreos et al., 2018]

Chapter 2

Background

We first provide the necessary background for the three classes of data structures we investigate in detail in this thesis. In brief, we cover hash functions, hash table, filters, and scan based structures.

2.1 Hash Functions

Hash functions are amongst the most powerful concepts in computer science as they provide a way to take arbitrary items and transform them into uniform random variables. In doing so, they enable the creation of many data structures that have excellent guarantees regardless of the data fed in to the structure.

The standard model of a hash function is that it is a completely random function mapping a domain $\mathcal{X}$ to a codomain $\mathcal{Y}$ such that 1) every item $x \in \mathcal{X}$ has equal probability of having as output every item $y \in \mathcal{Y}$ and 2) for any $S \subseteq \mathcal{X}$, the $|S|$ random variables $\{h(x)\}_{x \in S}$ are independent. More intuitively, $\mathcal{Y}$ is usually the set $\{0, 1, \ldots, m-1\}$ and if we denote our random hash function by H, then our model says that for any set of n inputs $x_1, \ldots, x_n$ and n outputs $a_1, \ldots, a_n$, we have

$$\begin{aligned}\mathbb{P}(H(x_1) = a_1, \ldots, H(x_n) = a_n) &= \prod_{i=1}^{n} \mathbb{P}(H(x_i) = a_i) \\ &= (\frac{1}{m})^n\end{aligned}$$

While this model, usually referred to as an ideally random hash function, is useful for the analysis of data structures, it is not realistically possible to create ideally random hash functions. This leads to questions both about other models for hash functions and the related question of what should our hash functions actually compute in practice.

From the theoretical side, the foundational formulation of hashing occurs in Carter and Wegman's introduction of *universal hashing* [Carter and Wegman, 1977], wherein they introduce the idea of choosing a random hash function h from a larger family H such that $\forall x_1 \neq x_2$, $\mathbb{P}(h(x_1) = h(x_2)) \leq 1/|\mathcal{Y}|$. This directly allows for computational guarantees on important hash data structures such as separate chaining hash tables (see next subsection), showing that ideal randomness is not always required for good performance. Still, this is not enough randomness for many data structures [Pătraşcu and Thorup, 2010, Pagh et al., 2011], and so an expanded idea of hash randomness is k-*independence*, which is that for any set of k inputs $x_1, \ldots, x_k$, and k outputs $y_1, \ldots, y_k$, the probability of $\mathbb{P}(\cap_i h(x_i) = y_i) = m^{-k}$ [Wegman and Carter, 1981]. Given this notion of k-independence, there is then a large body of literature on how much independence is necessary for a given data structure (see Cohen and Kane [2009], Schmidt and Siegel [1990], Pătraşcu and Thorup [2010], Pagh et al. [2011] for examples), and on designing k-independent hash functions [Wegman and Carter, 1981, Zobrist, 1990, Patrascu and Thorup, 2011].

In practice, systems designers generally avoid k-independent hash functions as they are deemed too computationally expensive. Instead, they opt for hash functions which lack formal robustness guarantees but are faster to compute and empirically observed to act like ideally random hash functions [Appleby, b, Ramakrishna, 1988, 1989, Pagh and Rodler, 2004]. For instance, RocksDB uses xxHash [Collet], Google heavily uses CityHash, Wyhash, and FarmHash [Pike and Alakuijala, 2011, Wang et al., 2020, Pike and Alakuijala, 2014], and C++ compilers such as g++ often choose MurmurHash [GNU Compiler Collection, Appleby, a]. These hash function do have well established principles about their design, and often base much of their design off almost-universal hash functions such as Multilinear-Modular-Hashing or Non-linear Modular-Hashing [Halevi and Krawczyk, 1997, Black et al., 1999]. Never the less, the hash functions in question are also often known to have provable collisions (i.e. $x_1 \neq x_2$ but $h(x_1) = h(x_2)$) [Aumasson and Bernstein, 2012]. Thus, what seems to matter most is that the sets which generate many collisions for these hash functions are extremely arbitrary, and so while these hash functions are open to adversarial hash flooding attacks, on naturally occurring data sets their outputs appear identical to the outputs of ideally random hash functions.

Exactly why hash functions used in practice give results that look ideally random is still unknown, but one potential explanation for this phenomena comes from pseudorandomness. The essential idea is that if the data itself is random enough, then hash functions with weaker guarantees in terms of independence can be shown to perform in expectation nearly identically to those that are fully random. In particular, it has been proven that 2-independent hash functions act nearly identically to ideally random hash functions if data comes from a block source with enough Rényi Entropy [Mitzenmacher and Vadhan, 2008, Chung et al., 2013]. This connection between entropy and the necessary complexity of hash computation is connected to work in this thesis presented later, wherein datasets with more entropy can do less computation while computing hash functions.

2.2 Hash Tables

Hash tables are amongst the most common data structures in computer science and implement a map between keys stored in the table and their associated values. They generally need to support 2 main operations, inserting a new key and associated value into the table and retrieving the value for a key (or reporting that the key is not in the structure). Because of their importance, the amount of related work on hash tables is extremely vast, but we cover the most relevant canonical structures for hash tables here as well as the state-of-the-art implementation techniques used by hash tables in production.

2.2.1 Separate Chaining Hash Tables

Separate chaining hash tables are one of two canonical implementations of hash tables. A separate chaining hash table is stored containing m slots, where m is a configurable parameter of the table, and each slot contains a linked list of key-value pairs. Associated with a table having m slots is a hash function mapping keys to $\{0, 1, \ldots, m\}$.

To insert a key x with its associated value v, $h(x)$ is computed and the key-value pair (x, v) is

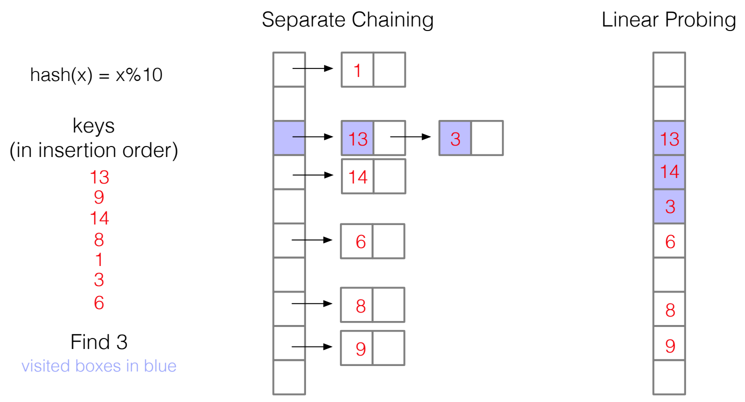


Figure 2.1: A depiction of Separate Chaining and Linear Probing Hash Tables (only keys shown)

put at the end of the linked list at slot $h(x)$. Similarly, looking for the value associated with a key x involves computing $h(x)$ and traversing the linked list at slot $h(x)$ until either the key-value pair containing x is found or the end of the list is reached (in which case x is not in the table). Figure 2.1 shows an example of a separate chaining hash table.

To analyze separate chaining, let X be a set of n keys inserted into the separate chaining hash table, and we will be interested in finding the mean number of comparisons needed to find a key $x \in X$. We will denote this random value by P, where though we are taking an average over the keys, the hash function is taken to be random.

Since swapping the insertion order of two keys does not change the average cost to access a key (for any hash function), we may take the insertion order of keys in X to be a random permutation. Now, examine a random $x \in X$. For each $x' \in P, x' \neq x$, it has probability $1/m$ of being in the same slot as x, and if in the same slot, a 50% chance of being before x (it was inserted before x) and a 50% chance of being after. Thus, a query for x will make $1 + \frac{n-1}{2m}$ comparisons in expectation, where the leading 1 comes from the fact a query for x will need to compare with itself with certainty. Since x was arbitrary, it follows that $E[P] = 1 + \frac{n-1}{2m}$. If we let $\alpha = \frac{n}{m}$ be the *fill* of the table, then $E[P] \approx 1 + \frac{\alpha}{2}$. Similar logic shows the average number of comparisons when looking for a key not in the table, denoted by $E[P']$, is $E[P'] = \alpha$.

Separate chaining hash tables are easy to manage and analyze because collisions only matter for the same slot, however compared to other hash tables they have poor data locality because of many pointer traversals and require extra space for the many pointers.

2.2.2 Linear Probing Hash Tables

Linear probing hash tables have great data locality properties and offer better performance than separate hash chaining tables on modern hardware. The majority of state-of-the-art hash tables in production today are variants of linear probing hash tables with additional optimizations on top (these optimizations are covered at the end of this subsection). From a theoretical side, linear probing hash tables are significantly more complex to analyze than separate chaining hash tables.

Linear probing hash tables start with a contiguous array of empty slots. To insert a key x into the hash table, the hash $h(x)$ is computed and used as an initial slot, and then the array is traversed in sequential order until either the key is found and its value replaced, or until an empty slot is found (the key is put there). Querying for a key follows the same approach and either returns the value associated with a key or that the key is not present. Figure 2.1 shows an example.

The first analysis of linear probing hash tables under ideally random hash functions was done by Donald Knuth [Knuth, 1963]. In the analysis, he shows that the average cost for querying a key in the table and the average cost for querying for a missing key satisfy:

$$E[P] = \frac{1}{2}(1 + Q_0(m, n-1)) \tag{2.1}$$

$$E[P'] = \frac{1}{2}(1 + Q_1(m, n)) \tag{2.2}$$

where $Q_i(m,n) = \sum_{k \geq 0} \binom{k+i}{i} \frac{n^{\underline{k}}}{m^k}$ and $x^{\underline{k}}$ represents x to the falling kth power. Using that $\frac{n^{\underline{k}}}{m^k} \leq \alpha^k$ gives that

$$E[P] \leq \frac{1}{2}(1 + \frac{1}{1-\alpha})$$

$$E[P'] \leq \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$$

A novel proof is given of these equations in Appendix A.

A line of work in theoretical computer science looks at what level of independence is required from a hash function to give similar performance to that of an ideally random hash function [Schmidt and Siegel, 1990, Pagh et al., 2011, Pătraşcu and Thorup, 2010]. Pagh et al. [2011] show that

5-independent hash functions suffice to give $O(\frac{1}{(1-\alpha)^2})$ performance and Pătraşcu and Thorup [2010] shows that using 4-independent hash functions can give logarithmic complexity in the time for linear probing. These results provide intuition on how well hash functions should need to perform in practice to give good results for linear probing hash tables; however, we note these proofs have much larger constant factors dropped by the Big-O notation and that equations 2.1 and 2.2 tend to be more predictive of performance in practice (for any reasonably well-designed hash function).

To optimize linear probing hash tables in practice, hash tables make use of *tag bits* which make comparisons between keys faster [Google, Bronson and Shi]. The general idea of tag bits are that they are short codes produced by a function on the inserted item. The tag bits are generally stored in a metadata array with the same number of slots as the linear probing array, and then when searching for a key, the table probes the metadata array (at the same slot values) to check if tag bits match and then only if so, goes on to check the array of keys. Tag bits tend to be 8 bits long exactly, so they take up one byte. This makes it so SIMD (single instruction multiple data) commands can compare the tag bits of the queried for item against the tag bits of many items in the table at once, making comparisons relatively cheap in modern hash tables. In terms of how tag bits are produced, tag bits are generally taken from the result of hash function evaluation. In particular, hash functions provide more random bits than are necessary for just inserting into a hash table (commonly hash functions provide 64 random bits), and so the leading bits are taken to form an item's tag whereas the other bits are used to get the address slot for the item.

2.3 Filter Data Structures

A filter data structure, also called an approximate membership query (AMQ) data structure, is a lossily compressed version of a set. In particular, this lossily compressed version of a set supports membership queries of the form "is x in my set?" and returns answers with no false negatives and a limited number of false positives. Matching the terminology of false negative and false positive, we will refer to items in the set being lossily compressed as positive elements and all other items as negative elements.

Filters have certain commonalities. First, all filters have a tradeoff between their two main goals of being small in memory footprint vs. producing as few false positives as possible. Second, all filter structures support at least the operations CONSTRUCT, which takes in a set S to represent lossily, and QUERY, which given an element x to query for, returns whether or not the filter believes the element is in the set. Additional operations that are commonly used, but not supported by all filters, are INSERT, the ability to insert new items to the filter, and DELETE, the ability to delete items from a filter. Some filters such as Bloom filters also support more complex operations like UNION-ing two filters to filters to create a filter which lossily represents the union of the two sets in the initial filters.

Filters fall into two broad categories: those that are workload-agnostic and provide the same false positive probability for all negatives, and those that are workload-dependent and which try to use either query skew or the semantics of what is included in the set to make different negative elements have different false positive probabilities.

2.3.1 Workload-Agnostic Filters

Workload-agnostic filters are the older, more commonly used form of creating filter data structures. They have the advantage that they can be constructed using only the set to be lossily represented, whereas workload-dependent filters will generally need a sample of past queries to be constructed. There are two main classes of workload-agnostic filters: Bloom filters and their variants, and fingerprint-based filters. We cover both briefly and refer readers to Broder et al. [2002] for more depth on Bloom filters and to the relevant cited papers for more information on fingerprint-based filters.

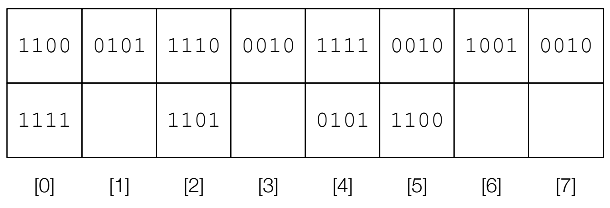
Bloom filters [Bloom, 1970] are initialized by setting an array of bits of size m to all zero. When adding an element, k independent hash functions $h_1(x), \ldots, h_k(x)$ are computed each giving results between 0 and $m - 1$. The bits bits at the locations $h_1(x), \ldots, h_k(x)$ are set to 1 (or kept at 1 if set already). Querying an element computes the same k hash functions and checks if each bit is set to 1; if all are, the filter returns the item is in the set whereas if any are 0 then it returns the item is not in the set. For an item in the set, this method guarantees the filter returns the item is in the set, whereas for items not in the set there is some probability that all bits were set 1 by items put into the filter. When optimally configured in terms of the number of hash functions, the false positive probability vs. size tradeoff for Bloom filters is approximately

$$\frac{m}{n} = -1.44 \log_2 \alpha$$

where n is the number of items stored and α is the false positive probability of the filter. See Broder et al. [2002] for details of the derivation. We note that this equation tells the space in bits per element needed to achieve a desired false positive probability. This is a commonality of all size vs. false positive probability equations for filters and so we will denote the bits per element needed by a filter (when optimally configured) to achieve a false positive probability of α to be $s(\alpha)$. Thus for a Bloom filter, $s(\alpha) = -1.44 \log_2 \alpha$.

Fingerprint-based filter structures work by storing a fingerprint for each key, where a key's fingerprint is the result of a hash function h computed on that key. The fingerprint is usually a fixed-width word containing some fixed number of bits k (k is tunable). Additional hash computation is used figure out how to store each fingerprint inside the data structure in way that tries to minimizes 1) the possible number of fingerprints to compare against and 2) the space overhead of storing all fingerprints.

A simple and non-optimal fingerprint filter is shown as an example in Figure 2.2. In the example, the filter contains 8 buckets each with two slots for fingerprints of bit-width 4. To store items, one hash function would be computed to figure out which slot an item goes into, and then the item's fingerprint would be stored in the first empty location of the bucket (or construction would fail if no empty locations exist in the bucket). Querying for an item would then follow the same steps and a positive would occur if either item in the bucket matches the fingerprint to be queried for. In our toy example, this means the false positive rate would be less than $2 * 2^{-4}$. Ideally, the goal of fingerprint filters is to make the filter have little empty space (with high probability of construction success) and



1100	0101	1110	0010	1111	0010	1001	0010
1111		1101		0101	1100		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Figure 2.2: Fingerprint filter

for queries to check against as few fingerprints as possible. The example given in Figure 2.2 is sub-optimal compared to state-of-the-art filters but shows all the relevant concepts.

More state-of-the-art examples of fingerprint-based filters use more sophisticated ways of storing fingerprints into the structure. For instance, Cuckoo and Morton filters are based on cuckoo-hashing ideas for hash tables [Fan et al., 2014, Breslow and Jayasena, 2018], quotient filters are based on space-efficient linear probing hash tables [Bender et al., 2012, Pandey et al., 2017], and xor filters exclusive-or together multiple items stored in a table to create the right fingerprint Graf and Lemire [2019] (based on ideas from Bloomier filters [Chazelle et al., 2004]).

To ensure a false positive probability for all negative items, it is required (for sets S taken from asymptotically large domains) that filters have $s(\alpha) \geq -log_2\alpha$ [Broder et al., 2002]. Both Bloom filters and fingerprint-based filters have size vs. false positive probability equations which are well approximated by equations of the form $s(\alpha) = \frac{-\log_2 \alpha}{f} + c$ with $f \geq 1, c \geq 0$. The goal is to make c close to 0 and f close to 1, with current approaches coming arbitrarily close to the lower bounds for space vs. false positive rate at the cost of higher construction times Dillinger and Walzer [2021].

2.3.2 Workload-Specific Filters

Workload-agnostic filters are defined by the fact that all negative elements have the same probability of being a false positive, which also implies the expected false positive rate (what proportion of queries at negative elements are expected to result in a false positives) for arbitrary workloads. If we care most about the expected false positive rate, and we know that certain negative elements are more likely to be queried or have information about how likely items are to be in the set, we can move beyond the theoretical lower bound for workloads implied by workload-agnostic filters and achieve lower false positive rates for specific workloads of interest. This is the approach taken by learned approaches to filtering, where machine learning models can provide us with their estimates of how likely an item is to be in the set stored and how likely items are to be queried. For instance, in the task of URL Blacklisting, a classifier may give us info about how likely a queried for item is to be in a list of spam URLs based off the semantics of the URL itself.

This side information is then used to treat elements differently based off the predictions of the classifier, with the main idea that items that are less likely to be in the set or have higher query frequencies should be given more stringent checks by the filter. For example, in weighted Bloom filters and their variants [Bruck et al., 2006, Wang et al., 2015, Dai and Shrivastava, 2020], items which have lower likelihood to be in the set compute more hash functions than those likely to be in the set. If the classifier is accurate, this means the filter itself will have fewer set bits and queries for negative elements will compute more hash functions, both lowering false positive rates. Other learning-based approaches differ in how they use information from the classifier [Kraska et al., 2018, Mitzenmacher, 2018, Vaidya et al., 2020], but have many of the same benefits. The drawbacks for all learning-based approaches are usually the same: in exchange for their better space vs. false positive rate trade-offs, they have to deal with the computational expense of the model and robustness concerns brought about by concept drift.

2.4 Database Scans: Predicate Evaluation over Unordered Arrays

We are going to be interested in evaluating equality predicates (find me values that equal x) and range predicates (find me values greater than x_1 and less than x_2) over arrays that are not sorted by value. The first thing to cover is why this problem formulation, and why is it central to databases, as the natural tendency to answer these predicates efficiently would be to perform some form of ordering or partitioning on the array of data. To get there, we start by reviewing the relational data model.

2.4.1 From Predicate Evaluation to Scans over Unordered Arrays

Relational database tables hold tuples, which consist of connected attribute values, and analytical relational databases desire fast evaluation of queries with predicates over different attributes for each tuple (i.e. find me all tuples such that attribute $A < 5$ and attribute $B < 10$). For single tables, there are usually many attributes to store, and queries often perform predicate evaluation over many of these attributes.

One central question in relational databases is then how to organize data to fast answer these queries. For instance, if a single attribute is used in almost all queries and helpful in selecting data, it makes sense to order or partition the whole table by that attribute, creating what is known as a primary index or a clustered index. All other attributes then follow the order of the chosen attribute. The limitation of this approach is that different primary key column requires a separate copy of the data (as only one ordering can be applied). While approaches to keeping multiple copies of the data have been tried [Abadi et al., 2006, 2013], it is common mostly to have only a single primary key and copy of the data, meaning data values and data ordering are unrelated for the majority of attributes.

A second major approach to evaluating predicates over unsorted data comes in the form of unclustered indexes. Here we induce some ordering or partitioning of the data in an auxiliary structure, and then this structure points back to the main copy of the data where other attributes can be retrieved. Unclustered indexes are usually variants of B-trees [Comer, 1979, Graefe, 2011] or hash indexes Ramakrishna [1988], Ramakrishnan and Gehrke [2002]. While this generally makes evaluation of the predicate much faster, the indirection caused by pointers back into the data and the complications arising from this often make overall query execution slower if enough data satisfies the predicate [Selinger et al., 1979]. This is known as the selectivity crossover point, and with modern analytical databases, the crossover point is as low as 1% [Kester et al., 2017].

Thus, as a result, we end up with the question of how to best optimize scans over unordered copies of data values and apply predicates in an efficient manner. A first technique that helps in this immensely is to partition the data by attribute, creating what are known as column-stores [Abadi et al., 2013], and read only the attributes needed for each predicate before combining the results. This reduces the problem to the one we study in detail: that of performing predicate evaluation over a single array of values from a specific ordered domain.

2.4.2 Techniques in Optimizing Scans over Unordered Arrays

The most basic approach to evaluating a predicate over an unsorted array of values is a simple in-order traversal over the data with an equality predicate ($A = 5$) or range predicate ($5 < A < 10$) applied to each value. While simple, this approach is hyper-optimized in analytical column stores. It is easily parallelized to many cores, increasing concurrency, and within a core extensive use of

SIMD commands are used to evaluate the predicate over many data items with just a single CPU instruction. As a result, this approach is fairly efficient, and is used in practice today in many query plans.

Several optimizations are added on top of this basic scan to improve performance. These include techniques using lightweight metadata about groups of values, early pruning techniques which partition single data values into disjoint sub-values, and lightweight compression techniques. We go through each below, explaining how they reduce data movement which is generally the main bottleneck for database scans.

Lightweight metadata techniques largely work as a way to skip data while doing an in-order scan. Zone Maps are amongst the most widely used techniques today, and work by storing small amounts of metadata such as min and max for blocks of data [Moerkotte, 1998]. This small amount of metadata exploits natural clustering properties in data and allows scans to skip over blocks that either entirely qualify or entirely do not qualify. Other techniques such as Column Imprints [Sidirourgos and Kersten, 2013] or Feature Based Data Skipping [Sun et al., 2014] take more sophisticated approaches, but the high level idea is the same: they use summary statistics over groups of data to enable data-skipping. These approaches take advantage of the clustering that tends to occur naturally in database attribute values, producing large benefits when they apply. However, they produce no benefits when data is not clustered [Qin and Idreos, 2016].

Early pruning methods such as Byte-Slicing [Feng et al., 2015], Bit-Slicing [Li and Patel, 2013, O'Neil and Quass, 1997], and Approximate and Refine [Pirk et al., 2014] techniques work by bitwise-decomposing data elements. On a physical level, this means partitioning single values into multiple sub-values, either along each bit [Li and Patel, 2013], each byte [Feng et al., 2015], or along arbitrary boundaries [Pirk et al., 2014]. After physically partitioning the data, each technique takes a predicate over the value and decomposes the predicate into conjunctions of disjoint sub-predicates. As an example, checking whether a two byte numeric value equals 100 is equivalent to checking if the high order byte is equal to 0 and the lower order byte is equal to 100. After decomposing the predicate into disjoint parts, each technique evaluates the predicates in order of highest order bit(s) to lowest order bit(s), skipping predicate evaluation for predicates later in the evaluation order if groups of tuples in some block are all certain to have qualified or not qualified. Substantial amounts of data are skipped if the data in the high order bytes is informative, however these techniques can suffer under data skew.

Lightweight compression techniques reduce the time to scan unordered arrays by reducing the total size of data read. Because data movement is the bottleneck for database scans when compression is lightweight, this reduces overall time to evaluate predicates. If compression is too heavyweight, the CPU cost of decompression outweighs the benefits of reduced data movement. The major techniques which are used in databases include dictionary compression for categorical data (such as country or college attended) [Müller et al., 2014, Willhalm et al., 2009], and frame-of-reference(FOR) encoding, delta encoding, prefix suppression, and null supression for numerical data [Zukowski et al., 2005, Westmann et al., 2000, Fang et al., 2010]. As these techniques will be used in the creation of Column Sketches, which we introduce in Chapter 5, we differ their discussion until then.

Chapter 3

Cerebral Hashing: Entropy-Learned Hashing

This chapter introduces our first example of designing cerebral data structures, Entropy-Learned Hashing, which looks at how randomness inside of incoming data items can be used to create faster hash function evaluation while preserving the approximate uniformity of the hash function outputs. We start by discussing the many uses of hashing and reviewing the traditional approaches to hash function design before diving into the approach for Entropy-Learned Hashing.

3.1 Dataset Specific Hashing

3.1.1 Hash Functions and their many uses across Systems

Hashing is one of the core concepts in computer science; data structures and algorithms which use hashing exist in nearly every computer program. It's most ubiquitous use case, hash tables, is the standard way to access individual data items. They are used both for fast access to hot data in L1 cache across general purpose programs as well as for accessing colder data that lies outside of cache either in memory or on disk. For example, in relational database systems hash tables are used for joins and group by operations. Beyond hash tables, hashing is used in numerous other core parts of computer science such as filters [Bloom, 1970], data partitioning [Polychroniou and

Ross, 2014], load balancing [Mitzenmacher and Sinclair, 1996], and sketches [Broder, 1997, Flajolet et al., 2007]. As a result of their many and important use cases, hashing is not only central within relational databases [Ramakrishnan and Gehrke, 2002, Richter et al., 2015] but acts as a core component of systems across compilers [Zhu, 2019], file systems [ZFS, 2019, Steve Tunstall, 2017], gaming [Gregory, 2009], genomics [Mohamadi et al., 2016], and more.

Because hashing is so ubiquitous, it is a substantial portion of overall system cost. Google states that 2% of its total CPU usage and 5% of its total RAM at the company is spent on just one hash-based data structure, hash tables, in just one of the languages used, C++ [Kulukundis]. Including other languages and other hash-based operations, the total CPU and memory usage spent on hashing overall is surely much higher. Meta makes similar statements, with developers stating that hash tables are such "a ubiquitous tool in computer science that even incremental improvements have large impact" [Bronson and Shi]. Moving from large cloud infrastructure to particular applications, inside databases hash-based joins and aggregations are amongst the most expensive and used operators; as a concrete example they account for over 50% of total time on 17 of the 22 queries on the TPC-H benchmark for Hyrise [Sirin and Ailamaki, 2020, Dreseler et al., 2020]. Another example can be seen in compilers, where using hash tables in linking is a substantial part of program compilation costs in Visual Studio [Zhu, 2019]. In addition to hash tables, hash-based filters are a core component of LSM-tree based key-value stores [Idreos and Callaghan, 2020] and can be a core computational bottleneck [Dayan and Twitto, 2021, Zhu et al., 2021]. Similarly, hash-based sketches act as a key computational bottleneck in network switches [Liu et al., 2019]. These observations across diverse industries, systems and data structures demonstrate that:

Despite numerous algorithmic and engineering advances, hash-based operations are still expensive because of the frequency and scale at which they are used.

3.1.2 Traditional Approaches to Hashing

A core component of all hash-based data structures and algorithms is the hash function itself. Hash functions have two primary goals. The first is to create uniformly random outputs for any number of input items. That is, the output should be jointly uniform as well as marginally uniform.

The second is computational efficiency. While ideally both goals would be optimally achievable, they are practically at odds with each other. Thus a central question is how much randomness is needed from the hash function for the operation at hand.

The traditional approach to this problem is to design hash functions which provide enough randomness regardless of input data. This holds across both theory and practice. Theory approaches work to design k-independent hash functions and analyze how much independence is needed for given data structures, whereas approaches used in practice try to design efficient hash functions which provide results which appear identical to ideally random hash functions on large benchmark of testing inputs. Both approaches are discussed in more detail in Chapter 2.1.

We focus on two aspects of the traditional approaches to hashing. The first is that while theoretical computer science can provide performance guarantees for data structures through uses of specific hash families, engineers in practice choose hash functions without robustness guarantees. This is because 1) the computational performance of hashing is too important, and 2) the outputs of these fast hash functions appear as random as that of an ideally random hash function [Appleby, b, Ramakrishna, 1988, 1989, Pagh and Rodler, 2004]. This emphasis of hash performance over robustness gives motivation to the problem that hash function computational speed is of large importance. The second aspect we focus on is that the attempt to make hash functions which work regardless of data inputs affects how these hash functions must work computationally. In particular, this approach implies that hash functions need to operate over every input byte. We focus on this specifically in Entropy-Learned Hashing, showing that if we have some structure in data inputs, context-specific hash functions can provide similar guarantees to traditional hashing without looking at every byte of data.

As mentioned in Chapter 2.1, one attempt to explain why empirically hash functions have outputs that appear identically to ideally random hash functions comes from pseudorandomness: if *data itself is random enough*, then hash functions with weaker guarantees in terms of independence can be shown to perform in expectation nearly identically to those that are fully random [Mitzenmacher and Vadhan, 2008, Chung et al., 2013]. This approach ties into our own approach: as data becomes more random, producing (approximately) uniform outputs requires fewer input bytes of data.

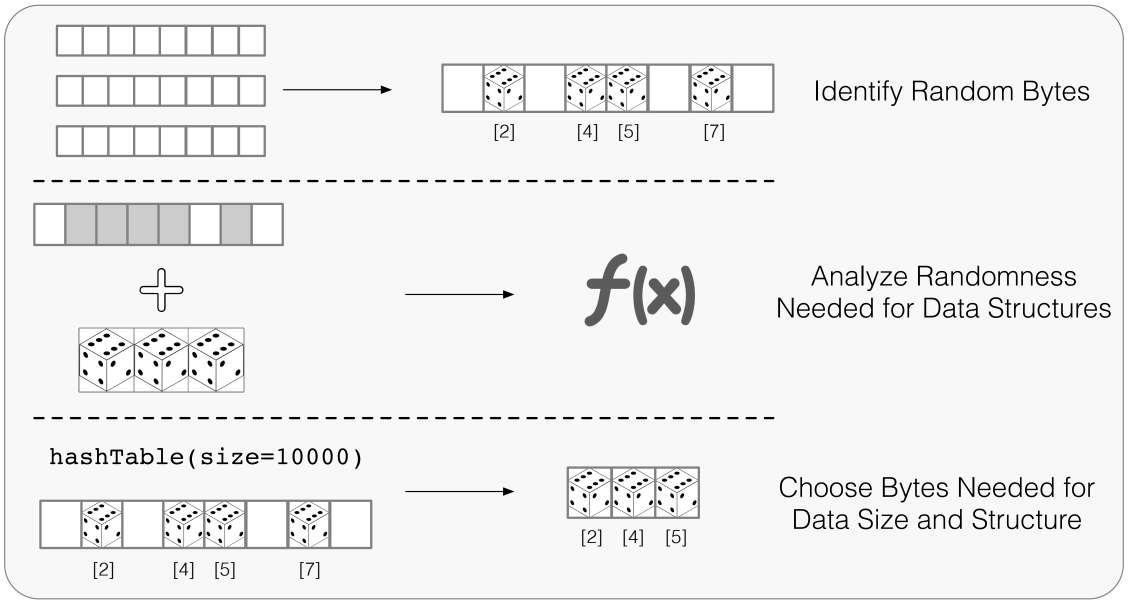
3.1.3 Entropy-Learned Hashing

We now use our framework of Cerebral Data Structures to improve hash function performance in a context-specific manner. In the previous paragraph, we motivated that for many hash-based data structures, the speed of its hash function as well as its uniformity is paramount to importance (step 1). We additionally noted that traditional hash functions which need to work for all data inputs must compute over every byte of input data (step 2a).

We now make use of our intuition for hashing, that if we know the randomness in input keys, we can use this to break many of the requirements of traditional hash functions which need to work over arbitrary inputs. In particular, we make use of the idea that if we take subsets of bytes which are random from many keys, the resulting sampled set of keys is likely mostly unique (with only a few duplicates). Then applying hash functions to this set of much smaller keys should produce outputs which are still nearly as uniform as traditional hashing while being significantly faster to compute. For example, for a dataset with keys of length 120 bytes, if a consistent subset of bytes (e.g., bytes 3,7,9,12, and 15) is sufficiently random, almost all keys will be unique using just these 5 bytes and so hashing only this set of five bytes would required only 1/24th the computation while the hash output is still close to uniformly random.

The rest of this chapter is then used to formalize this intuition and build out the components needed to turn this intuition into a data structure design. In particular, we find a new parametric quality of the workload (the Rényi entropy of incoming items) which can be translated into concrete metrics for several data structures and algorithms of interest including hash tables, filter structures, and data partitioning (step 2b). We show as well how to estimate this parameter of Rényi entropy given samples of past data items (step 3), and how to choose how many bytes are needed for hashing given our metric equations given parameter values and estimates of the parameter values (step 4). Figure 3.1 shows the overall process. Overall, this makes it so that we can go from samples of past data items to concretely realized hash functions which bring performance benefits on the task at hand.

The major breakthrough of Entropy-Learned Hashing is that the time it takes to hash input keys is now dependent on how random data is. As we show in our experiments, most datasets



Identify Random Bytes
[2]
[4]
[5]
[7]
Analyze Randomness Needed for Data Structures
f(x)
hashTable(size=10000)
Choose Bytes Needed for Data Size and Structure

Figure 3.1: The core steps in Entropy-Learned Hashing.

possess enough randomness that just a small number of bytes of data are needed for hashing for most tasks. This makes hashing time essentially constant and divorces its computation time from key size, as adding more bytes to a key no longer adds to hash computation time. This contrasts with traditional hash functions which have linear computational cost with key size. As a result of this difference, Entropy-Learned Hashing provides unbounded computational speedups as key sizes grow.

The major contributions of the Cerebral Data Structures approach to hashing are:

- *Entropy-Learned Hashing Formalization*: We introduce a new way to design hash functions that uses the entropy inside the data source to reduce the computation required by hash functions.
- *Optimization*: We show how to choose which bytes to hash given a collection of past queries and data items to analyze.
- *Generalization*: We show how the entropy of partial-key hashes generalizes to data items outside the given sample of data.
- *Concrete Trade-offs*: We derive metric equations for three core use cases of Entropy-Learned Hashing: hash tables, Bloom filters, and data partitioning. This allows trading speed in hash computation for small changes in other metrics of interest such as the number of comparisons,

FPR, and partition variance.

- *Computational Gains*: Comparing against state-of-the-art designs and implementations, Google's and Meta's hash tables, we show that Entropy-Learned Hashing provides higher throughput than traditional hashing. While this improvement is unbounded with respect to key size, for common medium-sized key types such as URLs, we show this improvement is up to 3.7× for hash tables, 4.0× for Bloom filters, and up to 14× for data partitioning.

3.2 Overview & Modeling

Having described the overall approach of Entropy-Learned Hashing, we now move on with a detailed description of the technique which will span the next three sections. In this section, we start with a more detailed overview as well as laying out the basics for notation and modeling which we use throughout the paper.

Overview. The goal of Entropy-Learned Hashing is to learn how much randomness is needed and to produce a hash function which does just enough work by controlling the input given to the hash function. To achieve this goal, Entropy-Learned Hashing looks for bytes which are highly random on input objects and passes just enough of these bytes to create a highly random output. Stated more formally, Entropy-Learned Hashing consists of creating a hash function H' which is the composition of 1) a partial-key function L which maps a key x to any subkey of x (including potentially the full key x), and 2) H, a traditional hash function. Our focus is on designing L, and H can be any of the many well-engineered hash functions for full-keys.

In order to create the partial-key function L, Entropy-Learned Hashing uses three steps as shown in Figure 3.1. First, it analyzes the data source x and identifies which bytes are highly random, and how much entropy can be expected from a choice of L (Section 3.3). Second, it reasons about how L affects data structure metrics (Section 3.4). Finally, it uses runtime information, such as the size of the desired Bloom filter or hash table or the number of partitions in partitioning to choose which bytes to use in L (Section 3.5).

Notation. The notation for all variables used is given in Table 3.1. Capital letters refer to either random variables or sets whereas lower case variables refer to fixed quantities. The new notation is

Notation	Definition (filter, hash table, or load balancer)
X, x	key stored in the filter or hash table
H, h	hash function for filter or hash table
Y, y	query key in filter or hash table
m	size of filter (in bits), table (in slots), or # bins
n	number of keys in filter or table
K	set of keys
$S_{\|L}$	multi-set of partial keys. Equal to $(K_{\|L}, z)$
$K_{\|L}$	Set of all partial keys.
z	maps each key $x \in K_{\|L}$ to $\|L^{-1}(x)\|$. z_x is used as shorthand for $z(x)$ throughout.

Notation	Definition (hash table only)
α	fill of hash table: $\frac{n}{m}$
P'	number of comparisons to find non-existing key
P	average # of comparisons to retrieve a key in the dataset

Table 3.1: Notation used throughout the paper.

because keys entered into H are no longer unique. The set of keys K contained in a hash-based data structure is broken down into the multi-set $S_{|L} = (K_{|L}, z)$. Here, K_L is the set of all partial-keys (outputs of L applied to keys in K), and z maps each key in K_L to the cardinality of its pre-image in K. For instance, if L takes the first two characters of an input and $K = \{\text{dog, dot, cat, fan}\}$, then $K_{|L} = \{\text{"do"}, \text{"ca"}, \text{"fa"}\}$, $z(\text{"ca"}) = 1$, and $z(\text{"do"}) = 2$.

Hash Function Model. We assume that H is ideally random, i.e. that for any distinct inputs $x_1, \ldots, x_n$, output range $[m] = \{1, \ldots, m\}$, and outputs $a_1, \ldots, a_n \in [m]$, we have

$$\begin{aligned}\mathbb{P}(H(x_1) = a_1, \ldots, H(x_n) = a_n) &= \prod_{i=1}^{n} \mathbb{P}(H(x_i) = a_i) \\ &= (\frac{1}{m})^n\end{aligned}$$

We do not use k-independent hashing; as noted before and as shown again in our experiments, hash functions tend to perform empirically like their perfectly random counterparts. Moreover, most proofs using k-independent hashing give big-O guarantees but drop constant factors [Pagh et al., 2011, Pătraşcu and Thorup, 2010, Mitzenmacher and Vadhan, 2008]. These constant factors are of significant importance for high performance hash functions.

Source Model. Conditioned on L we assume that the partial-keys $L(X)$ are i.i.d. distributed because the main metrics for hash-based algorithms tend to be order-independent. For instance, whether keys are ordered $x_1, \ldots, x_n$ or in the reverse order $x_n, \ldots, x_1$, the slots filled in a linear probing hash table or the length of the linked lists in a separate chaining hash table are identical. Similar statements hold for the false positive rate of Bloom filters and the partitions produced by partitioning. Thus, even if the original source has a temporal nature that might be better modelled by a Markovian assumption, the marginal distribution over time is more important.

3.3 Creating Partial-Key Functions

The first step is to create the partial-key function L which needs knowledge about the data we expect. In the case of fixed datasets, such as read-only indexes like those used in the levels of LSM-based key-value stores [O'Neil et al., 1996], this is the actual dataset. With updates, we need a sample of past data and queries.

Metric for Partial-Key Hash Functions. Partial-key functions have two metrics. The first is the number of bytes in their output, with fewer being better so that subsequent hash computation is faster. The second is the Rényi Entropy of order 2 of their output, also known as the collision entropy. For a given discrete random variable X, its Rényi Entropy of order 2 is $H_2(X) = -\log \sum_{i=1}^{n} p_i^2$ where p_i is the probability that X takes on the ith symbol in an alphabet $\mathcal{A} = \{s_1, \ldots s_n\}$. It draws its name from the fact that if X_1, X_2 are drawn i.i.d. from the same distribution as X, then $H_2(X) = -\log_2 \mathbb{P}(X_1 = X_2)$. We use collision probability to refer to $\mathcal{P}(X) = \mathbb{P}(X_1 = X_2)$ and mean Rényi Entropy of order 2 whenever we use the term entropy. For Entropy-Learned Hashing, Rényi Entropy tells us how likely collisions are to occur. The following lemma will be useful in our analysis:

Lemma 3.3.1. *Given n i.i.d. samples from a distribution X, the number of observed collisions over the number of 2-combinations is an unbiased estimator of the collision probability for X. That is, if n_i is the number of times a symbol s_i appears in the sample, then we have*

$$\mathbb{E}[\sum_i \frac{n_i^2}{2}] = \frac{n^2}{2}\mathcal{P}(X)$$

where $x^{\underline{2}} = x(x-1)$ *is the 2nd falling power. Equivalently,*

$$\mathbb{E}[\sum_i \frac{n_i^{\underline{2}}}{2}] = \frac{n^{\underline{2}}}{2} 2^{-H_2(X)}$$

Proof. There are $\binom{n}{2}$ possible 2-combinations in n samples, each of which can produce a collision. The probability of collision is $2^{-H_2(X)}$ and so the expected number of collisions is $\binom{n}{2}\mathcal{P}(X)$. □

Optimization.: Selecting the Bytes to Hash The goal is to optimize our two metrics on our optimization set, which is either the fixed dataset or a training set of a sample of prior data items. Since our two metrics are at odds, the goal is to find an optimal Pareto frontier establishing for each $k = 1, 2, 3, \ldots$, what set of k bytes from our full-key input produces the most entropy.

Insight into this problem, as well as potential solutions, can be found by analyzing the similar problem for maximizing Shannon entropy (equivalently, Rényi entropy of order 1). In particular, for Shannon entropy selecting the best subset of size k of random variables from amongst n random variables is known to be NP-hard [Ko et al., 1995], suggesting that an optimal solution for Rényi entropy is likely computationally difficult. However, the greedy algorithm, described in detail below, is known to provide a $1 - \frac{1}{e}$ approximation to the best possible solution for Shannon Entropy because Shannon Entropy is submodular [Nemhauser et al., 1978]. Additionally, real-life applications of the greedy algorithm tend to get solutions which are close to the optimal solution [Balkanski et al., 2021]. Inspired by this success and by the connectedness of Rényi and Shannon entropy, we use the greedy algorithm to optimize Rényi entropy on our training set.

We start by using a dummy hash which reads zero bytes of the data items. Then, we continually add new bytes to the partial key function L in a way that decreases the number of collisions the most on the training data. After each new chunk of bytes, we record the entropy (either on the fixed dataset or on a validation dataset if data is not fixed) and repeat the process. We stop when L has no collisions on the training data, and note that at each iteration of the algorithm we need only to look at data items which are not unique given previous bytes chosen for L, reducing algorithm runtime substantially (items that are not equal on a subset of bytes cannot be equal on a larger subset). At the end, we have a sequence of partial-key functions which are our solutions

Algorithm 1 ChooseBytes

```
Input: train_data: either data items or sample of past data items
Input: test_data: data to check entropy on (if not for fixed dataset)
1: positions = vector()
2: entropies = vector()
3: max_len = maximum length of any data item
4: while not all partial keys unique do
5:     positions.push_back(NextByte(data,max_len,positions))
6:     entropies.push_back(EstimateEntropy(test_data, positions))
7:     data = NonUnique(data, positions)
8: return positions, entropies
```

for $k = 1, 2, 3, \ldots$ bytes, with higher k meaning more input bytes are read but also monotonically increasing the entropy of the output.

Algorithms 1 and 2 give (simplified) pseudocode for this procedure. Additionally, Figure 3.3 shows example output from the procedure. While for simplicity Algorithm 1 is shown choosing 1 byte at a time, our implementation chooses 4 or 8 bytes at a time. This is because most modern hash functions which come after L operate one word of data at a time. In addition, we limit the maximum byte being chosen for partial-key hashing so that 90% of data items are under that data size. In the end, H' looks as follows:

```
if len(x) > last byte used in L:
    return H(L(x))
else
    return H(x)
```

Because we designed L so that almost all keys satisfy the first if statement, this makes the full hash function have predictable branching statements. This initial if statement is also dropped if the keys are of fixed length. The result, when L is tightly integrated into the hash function H, is that H' has predictable branches and a small instruction count on average.

Evaluating the Resulting Entropy. To make decisions on how many bytes are needed, we need an estimate of the entropy of $L(X)$. When data is fixed, we use the training set as a ground truth value for the entropy. When generalization to new data is needed, we use separate validation data.

To estimate the entropy of $L(X)$, we compute the empirical collision probability on the validation set V by 1) computing $L(x)$ for each x in V, 2) counting the number of collisions, and then 3) dividing this by $\frac{v^2}{2}$ where v is the number of items in V. From Lemma 3.3.1, this gives an unbiased estimate of the collision probability. To get an estimate $\hat{H}_2$ of the entropy, we take the negative log

Algorithm 2 NextByte

```
Input: data: either data items or sample of past data items
Input: max_len: maximum length item in dataset
Input: past_bytes: past bytes chosen
 1: min_coll, min_i = $\infty, -1$ // track of min # collisions, most entropic byte
 2: for $i = 0$ to max_len do
 3:     count_table, num_coll = $\{\}, 0$
 4:     for $j = 0$ to len(data) do
 5:         p_key = $data[j]$ using (past_bytes, i) // form partial-key
 6:         p_key = (len(data[j]), p_key) // length is always part of partial-key
 7:         count_table[p_key] + = 1 // increment count partial-key
 8:         num_coll += (count_table[p_key] - 1) // add collisions (if any)
 9:         if num_coll < min_coll then
10:             min_coll, min_i = num_coll, i // update best byte
11: return min_coll, min_i
```

of this number.

Given this estimator, the natural question to ask is "how many samples are needed?". The techniques of [Acharya et al., 2017, Obremski and Skorski, 2017] use the birthday paradox to answer this question; namely, if we want to say that the entropy is at least some value H_2 with confidence, we need $O(2^{H_2/2})$ samples. As we will show in Section 3.4, data structures or algorithms storing n elements will generally need H_2 to grow at a rate of $\log_2 n$, suggesting $O(n^{1/2})$ samples is enough to say with probability approaching 1 whether or not $L(X)$ has enough entropy for a given task. Giving a concrete example, when using v validation samples a 99% confidence estimator for the entropy is: $H_2 \geq \min \begin{cases} \hat{H}_2 - 2 \\ \log_2 \frac{v^2}{400^2} \end{cases}$ with probability 0.99. Thus if our data structure needs entropy $H_2 = \log_2 n$, setting $v > 400\sqrt{n}$ is enough validation samples to say with high probability whether or not $L(X)$ has the required entropy [1].

The most important takeaways are that the number of validation samples needed both varies with the data size and also grows slowly with the data size. Thus, when we want to use Entropy-Learned hashing on small data, the sample can be small because we only need to make sure it has just enough entropy. When the data is large, the number of samples needed grows but much more slowly than the data size.

[1]We note this leading constant seems higher than what is necessary in practice, suggesting possible further improvements in the analysis of this estimator.

3.4 Connecting Entropy to Data Structure Performance

The next step in Entropy-Learned Hashing is understanding the entropy needed for a given system task, i.e., a data structure or algorithm used in a system. We study specifically the entropy needed by three of the mostly widely used hashing tasks, namely:

1. **Hash tables** which are the default way to access data by equality, and which are widely used across general purpose programs including relational systems and key-value stores.
2. **Bloom filters** which are used to reduce accesses to a set and are used in databases to reduce the costs of joins in OLAP systems as well as point queries in key-value stores.
3. **Partitioning** which is a core step in numerous algorithms.

Each of these tasks has multiple metrics of interest, including: CPU cost, memory footprint, throughput, false positive rate, and much more. The three hash-based operations above present a diverse set of expressions of these metrics. For example, Bloom filters have small memory footprint compared to the other components, while they all have drastically different characteristics in terms of output write patterns which affects the overall throughput.

By creating cheaper to compute hash functions we improve the computational efficiency; what is left to show is that the small increase in expected collision probability does not result in significant degradation on other metrics. For hash tables, the metric of interest for performance is the number of comparisons needed to retrieve a key. For Bloom filters, it is the false positive rate and for Partitioning the variance of the distribution of data amongst bins.

There are two takeaways from the analysis in this section. The first is that we can argue formally about the needed entropy from partial-keys for data structures to behave as desired. This allows us to design Entropy-Learned hash functions which bring end-to-end performance benefits. Second, the analysis shows that across all tasks, Hash tables, Bloom filters, and Partitioning, the needed amount of Renyi entropy in $L(X)$ is approximately $\log_2 n$ plus a constant c. Thus, for a fixed dataset size, hashing needs only a constant amount of randomness. If some fixed set of bytes provides this amount of randomness, then hashing only need look at these bytes and its computation becomes independent of key size in that adding more bytes to the key does not increase hash computation time. Additionally, the dependence on n reaffirms our central thesis and further clarifies where Entropy-Learned Hashing is most useful: for large (hence random) objects or small datasets state-of-the-art hash functions do more work than necessary. The value of c depends on how much collisions affect a data structure; for instance, hash collisions in Bloom filters produce a certain false positive and so this has a high value of c, whereas for hash tables a collision produces an extra comparison which is more tolerable and so c is lower.

3.4.1 Hash Tables

Two prototypical designs of hash tables are separate chaining and linear probing [Cormen et al., 2009]. Separate chaining stores an array of linked lists. To query for an item, separate chaining hash tables 1) perform a hash calculation to get a slot a between 0 and $m-1$ and then 2) traverse the linked list at slot a until either the key is found or the end of the list is reached (the key is not present). Linear probing stores an array of keys and queries the table by 1) performing a hash calculation to get an initial slot a, and then 2) traversing the array in sequential order until either the key is found, or until an empty slot is found (the key is not present). Separate chaining tables are easier to manage and analyze because collisions only matter for the same slot, however they have poor data locality because of many pointer traversals and require extra space for the many pointers. In contrast, linear probing offers better performance but is more difficult to analyze and manage because of complex dependencies between hash values.

3.4.1.1 Hash Tables: Separate Chaining

Fixed Data. We first analyze separate chaining hash tables when the data is known which is an important class of indexed data. We then show this analysis translates from known data to random data.

Given $S_{|L} = (K_{|L}, z)$, when querying for an item y not in K, the expected number of comparisons P' is

$$\mathbb{E}[P'|y] = z_y + \frac{n - z_y}{m} \approx z_y + \alpha$$

This is because the (likely 0) z_y items which have the same partial key for sure are in the same slot, and the other $n - z_y$ items have $1/m$ chance of being in the same slot. This cost of querying for a missing key is also equal to the cost of adding a new item into the hash table, and this relationship holds true for linear probing as well. This is because additions first verify the item is missing and then put the item into the first empty slot they find.

By the same logic, querying for a key x in K costs $1 + \frac{1}{2}(z_x - 1 + \frac{n-z_x}{m})$ comparisons on average. The leading 1 is because the query key for sure compares with itself, and the second term is 1/2 times the expected number of items in the same slot as x. Summing across all data, the average cost P of querying for a key satisfies:

$$\mathbb{E}[P] \leq 1 + \frac{1}{2}\alpha + \frac{1}{2}\sum_{x \in K_{|L}} \frac{z_x^2}{n}$$

Random Data. When generalizing partial-key hashing to unseen (random) data, the above equations can be viewed as conditional expectations where we condition on the data. By using Adam's Law, i.e. $E[X] = E[E[X|Y]]$, we can average over the possible produced datasets given by the random data. Using the union bound and Lemma 3.3.1, the expected cost of querying for a missing key and the average cost for querying for a key satisfy

$$\mathbb{E}[P'] \leq \alpha + n2^{-H_2(L(X))} \tag{3.1}$$

$$\mathbb{E}[P] \leq 1 + \frac{1}{2}\alpha + \frac{1}{2}(n-1)2^{-H_2(L(X))} \tag{3.2}$$

Comparison with Full-Key Hashing. For full key hashing, the corresponding costs for querying for a missing key and the average cost to query for a key are

$$\mathbb{E}[P'] = \alpha$$

$$\mathbb{E}[P] = 1 + \frac{1}{2}\frac{n-1}{m} \approx 1 + \frac{1}{2}\alpha$$

This shows the tradeoff between partial key hashing and full key hashing. The number of comparisons is lower for full-key hashing, but this advantage goes exponentially fast to 0 as the entropy of the partial key hash increases. At the same time, the partial-key hash is significantly cheaper to compute.

Looking at the required relationship between n and the needed entropy of the input sub-keys further clarifies when and why partial-key hashing is useful. When $H_2(L(X)) > \log_2 n$, the number of extra comparisons needed drops below 1 and continues to drop exponentially fast with more entropy. Since hashing objects is more expensive than comparing them, this point represents near definite savings; the hash computation for the table is much faster while the work after the hash function is nearly the same.

3.4.1.2 Hash Tables: Linear Probing

Because of the complex dependencies between hash values and collisions, linear probing is significantly more complicated to analyze resulting in lengthier proofs. We provide a high level overview of the results here while detailed proofs can be found in Appendix A. We start with full-key hashing. We analyze the expected length of a full chain T for a new item added to the hash table. The chain includes the empty position on a chain's right side but not on its left side. Figure 3.2 shows an example.

Full-Key Hashing. In Appendix A, we provide a novel analysis of linear probing showing that the expected length of T satisfies $E[T] = Q_1(m, n)$ where $x^{\underline{k}}$ is the k-th falling power and $Q_i(m, n) = \sum_{k \geq 0} \binom{k+i}{i} \frac{n^{\underline{k}}}{m^k}$. For a new item, each location in a probe chain is equally likely as a hash location and so the expected probe cost given T is $E[P'|T] = \frac{1}{2} + \frac{1}{2}T$. Using Adam's law, it follows that

$$E[P'] = \frac{1}{2}(1 + Q_1(m, n)) \leq \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$$

which matches the known equations given by Knuth in [Knuth, 1998].

The average cost to query a key is then equal to the average cost to insert each key. Since the insertion cost $E[P']$ depends on n, we use P'_i to denote the cost when there are i keys in the table.

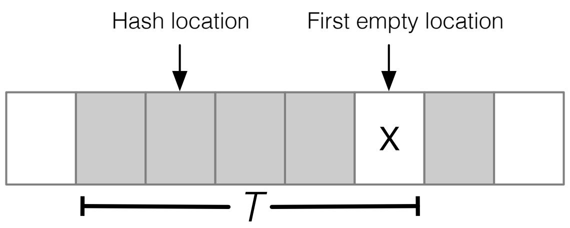


Figure 3.2: Example of a linear probing chain.

The average cost to query a key is then

$$E[P] = \frac{1}{n}\sum_{i=0}^{n-1} E[P'_i] = \frac{1}{2}(1 + Q_0(m, n-1)) \leq \frac{1}{2}(1 + \frac{1}{1-\alpha})$$

Partial-Key Hashing: Fixed Data. When given $S_{|L} = (K_{|L}, z)$, the expected length of the probe chain T depends on the number of partial key matches for the inserted key y, and satisfies

$$\begin{aligned} E[T] &\leq Q_1(m,n) + z_y Q_0(m,n) + \sum_{x \neq y} \frac{z_x^2}{m} Q_1(m,n) \\ &\leq \frac{1}{(1-\alpha)^2} + \frac{z_y}{1-\alpha} + \sum_{x \neq y} \frac{z_x^2}{m(1-\alpha)^2} \end{aligned}$$

When the new key is unique, the most common scenario when $H_2(L(X))$ is high, each location in the probe chain is equally likely and so $E[P'|T] = \frac{1}{2} + \frac{1}{2}T$. However, when the new key is not unique, each position in the chain is no longer equally likely. Thus we make the worst case assumption that it is at the end of the probe chain.

$$E[P'] \leq \begin{cases} \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2} + \sum_{x \neq y} \frac{z_x^2}{m(1-\alpha)^2}) & \text{if } z_y = 0 \\ \frac{z_y}{1-\alpha} + \frac{1}{(1-\alpha)^2} + \sum_{x \neq y} \frac{z_x^2}{m(1-\alpha)^2} & \text{if } z_y > 0 \end{cases} \tag{3.3}$$

When translating from P' to P, we again have that $E[P] = \sum_{i=0}^{n-1} E[P'_i]$. Since the cost of inserting each key is no longer the same, there is the question of how to evaluate this expression. Here, we make use of a fact first noticed in [Peterson, 1957], that the average cost of querying is equal for any order in which the items are inserted. Thus, in evaluating $E[P] = \sum_{i=0}^{n-1} E[P'_i]$, we may choose the insertion order of the items. Inserting all keys with non-unique partial-keys first and then inserting all keys with unique partial-keys gives the following bound for $E[P]$.

$$\begin{aligned} E[P] &\leq \frac{n-d}{2n} + \frac{1}{2}Q_0(m,n) + \frac{c}{m}Q_0(m,n) + \frac{c+d}{2n}Q_0(m,d) \\ &\approx \frac{1}{2}(1+\frac{1}{1-\alpha}) + \frac{c}{n} + \frac{c}{m}\frac{1}{1-\alpha} \\ &\leq (\frac{1}{2}+\frac{c}{n})(1+\frac{1}{1-\alpha}) \end{aligned} \tag{3.4}$$

We use $c = \sum_x z_x^2$ for the number of collisions and $d = \sum_{x:z_x \geq 2} z_x$ as the number of items that are duplicated keys. The above approximation assumes that d/m is small, which is the case whenever most keys are unique. This holds true with probability near 1 if entropy is sufficiently large.

Random Data. Using equations (3.3), (3.4), and Lemma 3.3.1 as well as Adam's Law, we have

$$E[P'] \leq \frac{1}{2}(1+\frac{1}{(1-\alpha)^2}) + n2^{-H_2(L(X))}\frac{3}{2(1-\alpha)^2} \tag{3.5}$$

$$E[P] \leq \frac{1}{2}(1+\frac{1}{1-\alpha}) + n2^{-H_2(L(X))}(1+\frac{1}{1-\alpha}) \tag{3.6}$$

Comparison With Full-Key Hashing. The tradeoffs between partial-key hashing and full-key hashing are similar to separate chaining. Again, we have a slight increase in comparisons as a tradeoff for significantly faster hash function evaluation. The expected number of comparisons again drops exponentially fast with the source entropy and $H_2(L(X))$ needs only to be in the same order of magnitude as $\log_2(n)$ for the extra needed comparisons to be small. Thus, as before, partial-key hashing makes the work of computing hash functions significantly cheaper while the work after the hash function is near identical, producing a net performance benefit.

3.4.2 Bloom Filters

For Bloom filters, the central trade-off is between the speed of the filter and the false positive rate (FPR) of the filter. As the number of bytes given as input to the hash becomes smaller, hashing becomes faster but there is a greater possibility of a partial-key collision, creating a certain false positive.

More formally, let $FPR(m,n,H)$ denote the false positive rate of a Bloom Filter using m bits, storing n items and using a hash function H. For a Bloom Filter using partial-key hash $H' = H \circ L$, its number of set bits is a function of the number of distinct items fed to H. If no keys collide on L, then it becomes a traditional Bloom Filter storing n items and using H. If there are $n' < n$ distinct items after L, then the resulting filter structure has the same number of set bits as one containing n' items. So for query key $y \notin K_L$, it has a false positive rate of $FPR(m,n',H)$, whereas if $y \in K_L$ it has a false positive rate of 1. It follows that our Bloom Filter using h' has exactly the following false positive rate:

$$FPR(m,n,H') = \mathbb{P}(Y_{|L} \in K_L) + FPR(m,n',H) \tag{3.7}$$

The second term is less than $FPR(m,n,H)$ as Bloom Filters' false positive rates increase with the number of items stored. If keys and non-keys are very different conditioned on the set of bytes L, then it is possible to make the FPR less than that of a standard Bloom filter by having $n' << n$ and $\mathbb{P}(Y_{|L} \in K_L) \approx 0$. However, we will generally ignore this case and focus on the case where keys and non-keys have the same distribution conditioned on L. In this case, a convenient bound for (3.7) is

$$FPR(m,n,H') \leq \mathbb{P}(Y_{|L} \in K_L) + FPR(m,n,H) \tag{3.8}$$

which is the FPR of a standard Bloom filter plus the probability that the query key matches some item in the key set on the bytes L.

Using the union bound, equation (3.8) translates to:

$$FPR(m,n,h') \leq n2^{-H_2(L(X))} + FPR(m,n,h) \tag{3.9}$$

Comparison With Full-Key Hashing. The above analysis reaffirms the central takeaway of our analysis of hash tables; the entropy of the dataset needs to be on the order of $\log_2 n$. For Bloom filters, a reasonable additional goal is that the increase in FPR be no more than some chosen ε. In this case, we need $H_2(L(X)) > \log_2 n + \log_2(1/\varepsilon)$. So an additional entropy term is needed to say that collisions are very rare for new partial-keys. As we show in our experiments, datasets often have this surplus entropy and so the Bloom Filter becomes significantly faster without suffering any false positive rate increase.

3.4.3 Partitioning & Load Balancing

With Partitioning the goal is to distribute n items, e.g., tuples or computational tasks, to a set of m bins. Here, we characterize how even this allocation is by analyzing the variance of the number of items assigned to each bin when each input key is unique. At lower variances, each bin is distributed closely around the average number of items n/m whereas higher variance suggests the bins are highly uneven. One important challenge comes when keys are skewed and heavy hitters exist. While challenging, the unevenness comes from the existence of heavy hitters rather than the quality of the hash function, and so we focus on the hash quality by considering the partitioning of all unique items.

Full-Key Hashing. With full-key hashing, the variance of each bin is the variance of a binomial with n balls each with probability $1/m$. Thus for a specific bin, its number of assigned objects Y has $Var(Y) = \frac{n}{m} - \frac{n}{m^2}$.

Partial-Key Hashing: Fixed Data. The probability of each key in K_L being assigned to a specific bin is distributed as an independent Bernoulli trial with probability $\frac{1}{m}$. Letting $\mathbf{1}_{H(x)=i}$ be the event that x was hashed to bin i, the variance of the number of objects Y assigned to bin i is

$$Var(Y|K_{|L}) = Var(\sum_{x \in K_{|L}} z_x \mathbf{1}_{H(x)=i}) = (n + \sum_{x \in K_{|L}} z_x^2)(\frac{1}{m} - \frac{1}{m^2})$$

Partial-Key Hashing: Random Data. For random data, we use the same conditioning arguments as before. Using Eve's Law, i.e. $Var(Y) = \mathbb{E}[Var(Y|K_L)] + Var(\mathbb{E}[Y|K_L])$, we can calculate the variance on random data. First, we note that for any set K_L, the value of $E[Y|K_L]$ is n/m by the randomness of the hash function (each bin is equally likely to contain any item). Thus

$Var(\mathbb{E}[Y|K_L]) = 0$ and again using Lemma 3.3.1, we have

$$Var(Y) \leq (1 + n2^{-H_2(L(X))})(\frac{n}{m} - \frac{n}{m^2}) \tag{3.10}$$

Comparison With Full-Key Hashing. As before, $H_2 > \log_2 n$ is enough for partial-key hashing to have similar variance to full-key hashing in terms of absolute terms. Thus, as in prior cases, once $H_2 > \log_2 n$ we have faster computation in terms of partitioning without sacrificing on the quality of our partitioning.

An important secondary argument for load balancing is whether we care about the absolute deviation from the mean or the percentage deviation away from the mean. While the absolute variance grows with n, the relative standard deviation, i.e. the standard deviation over the mean, of the bins decreases with n so that it becomes less and less likely that some bin has $x\%$ more than its expectation. In particular, the relative standard deviation is less than

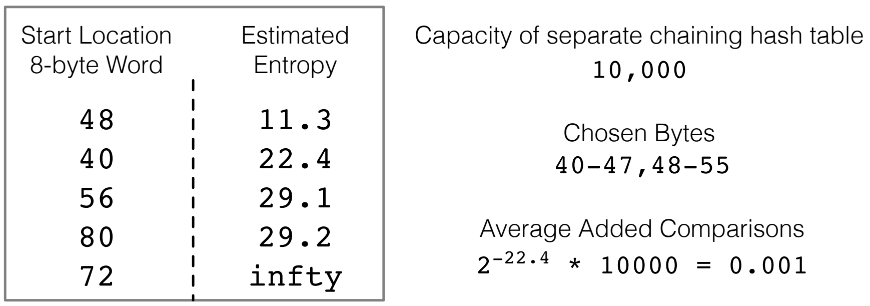
$$\sqrt{\frac{m}{n}}\sqrt{1 + n2^{-H_2(L(X))}} \approx \sqrt{m2^{-H_2(L(X))}} \tag{3.11}$$

Since the expected distance from the mean for a binomial is dominated by its standard deviation [Blyth, 1980], the above statement actually says that a bin's expected proportional deviation away from its mean is less than (3.11). So for instance, if we want a partition to be within 5% of its mean on average, we can achieve this by having $H_2 \geq 2\log_2 \frac{1}{0.05} + \log_2 m$.

Thus partitioning and load balancing have two regimes with regards to Entropy-Learned Hashing. When small absolute variance is required, we need $H_2(L(X)) > \log_2 n$; however, when n is large and we are simply interested in bins being relatively similar sizes, we can let $H_2(L(X))$ be greater than $\log_2 m$ plus a small constant, where the constant controls how much deviation is allowed.

3.5 Runtime Infrastructure

Section 3.3 showed how to estimate the entropy of datasets when conditioned on partial-keys and Section 3.4 showed how much entropy is needed for important hashing-based tasks. This



Start Location 8-byte Word	Estimated Entropy
48	11.3
40	22.4
56	29.1
80	29.2
72	infty

Capacity of separate chaining hash table
10,000

Chosen Bytes
40-47,48-55

Average Added Comparisons
$2^{-22.4}$ * 10000 = 0.001

Figure 3.3: The amount of bytes needed is based on the data and the current data structure capacity.

section brings everything together by explaining how to utilize Entropy-Learned Hashing at run time: namely, given a hash-based task and analysis of a dataset, choose the Entropy-Learned Hash function to have just enough randomness. Additionally, this section covers runtime infrastructure related to robustness so that Entropy-Learned Hashing retains the trustworthiness of traditional hash data structures.

Creating Hash Tables. Hash tables have a maximum capacity beyond which they need to rehash the stored items into a new larger table. This keeps the load factor low and therefore query times low. For Entropy-Learned Hashing, we use this maximum capacity before rehashing to decide L. In particular, for separate chaining hash tables, we choose L such that $H_2(L(X)) > \log_2 n + 1$, where n is the maximum number of items the current table will hold before rehashing. For linear probing hash tables, we choose L so that $H_2(L(X)) > \log_2 n + \log_2 5$. Both values are chosen based on the equations governing the number of comparisons, i.e. equations (3.1), (3.2), (3.3), and (3.4), and make sure the number of comparisons executed using partial-key hashing and full-key hashing are similar. An example of how the current capacity is used to choose L is shown in Figure 3.3, where an initial table with capacity 1000 uses just the 8-byte word at location 48 to hash keys.

As the capacity of a hash table changes (as new items are inserted), a rehash is triggered causing each item to be reinserted. Entropy-Learned Hash tables uses this opportunity to change the hash function; for instance, when key 1001 is inserted into the hash table from Figure 3.3, a rehash is triggered causing the table to grow. If the new capacity is above $2^{11.3} = 2521$, the partial-key function adds another word to increase entropy to the required amount. As a result, the hash table maintains just the right amount of entropy needed throughout its life cycle, using as cheap a hash function as possible without adding substantial extra collisions.

Bloom Filters. Bloom Filters need an estimate on the number of items they will hold before their creation. This is because, without access to their base items, they have no access to grow the number of bits being used. While there are techniques around this [Almeida et al., 2007], these come with space and computation tradeoffs and it remains true that standard Bloom filters need an up-front estimate of the number of data items. For Entropy-Learned Hashing, this makes it simple to choose the hash function. Given a maximum number of items n and an allowable added FPR of ε, we set the partial-key hash function to have entropy $H_2(L(X)) > \log_2 n + \log_2(1/\varepsilon)$.

Partitioning. For partitioning we require an estimate on the maximum number of items to be partitioned. We also need user input on how even they want partitions to be. If absolute variance is of primary importance (so that partitions are unlikely to vary by more than some # of tuples regardless of partition size), then setting $H_2(L(X)) > \log_2 n + c$ assures that variance is no more than $(1 + 2^{-c})$ times its usual amount. The default value of c which we use is 3. When relative variance is more important, and users need partitions to be roughly even (i.e. within 100c% of each other's size), we set $H_2(L(X)) > \log_2 m - 2\log_2 c$ as dictated by equation (3.11). We use $c = 0.05$ by default so that partitions are expected to be within 5% of their expected size.

Robustness. While Entropy-Learned Hashing makes only weak assumptions, namely that data which are somewhat random remain somewhat random, it recovers good performance quickly when assumptions are violated. Entropy-Learned Hashing is the most robust for hash tables. This is for multiple reasons, namely: 1) if collisions are as expected on items in the dataset, queries for both keys in the data and not in the data return quickly (Section 3.4), 2) hash tables can monitor collisions during insertions with little overhead, and 3) rehashing is an acceptable operation in hash tables by default (it occurs in all standard hash table libraries). This third point is the most key, and Entropy-Learned Hashing can rehash hash tables if collisions ever deviate from what is expected, falling back to full-key hashing if needed. For Bloom filters, their # of set bits concentrates sharply around their expected value [Broder et al., 2002], and this fact is used during construction of Entropy-Learned Bloom filters to validate that the data items fit the expected level of randomness. However, if they do not, or if queries are substantially different than the inserted items, the filter must be rebuilt. For partitioning, the cost of overloaded bins depends on the context, but for many contexts, such as in-memory radix partitioning, this can be solved by dividing the one or two overloaded bins into multiple bins.

3.6 Experimental Evaluation

We now demonstrate that, by identifying and utilizing surplus randomness in data, Entropy-Learned Hashing brings critical performance benefits against the top hash functions used at scale today by Google and Meta and across a diverse set of hash-based core components of modern systems.

Our experimental evaluation consists of three parts. The first part, which contains the bulk of our experiments, shows that Entropy-Learned Hashing produces sizable benefits of up to 3.7×, 4.0×, and 14× for common medium-sized key types such as URLs and text data. The second part of our experimental section covers benefits from Entropy-Learned Hashing on large keys such as those that would appear in deduplicating file blocks, with speedups of several orders of magnitude. Finally, we cover training time for Entropy-Learned Hashing and present the run times for applying the greedy algorithm to select bytes to hash.

3.6.1 Setup and Methodology

Data Structures and Operations. We use a diverse set of data structures and operations to apply Entropy-Learned Hashing: we test with Hash tables, Bloom filters, and Partitioning.

For **hash tables**, we compare against Google's hardware-efficient linear-probing hash table implementation, SwissTable [Google, Kulukundis]. This is the default hash table used in C++ throughout all Google operations, and has been heavily optimized as a result of the large computational footprint of hash tables at Google. A particular implementation note for SwissTable is that it first does linear probing into an array of tag bits (8 bits per key) to see if chosen bits from hash values match, and only if they do, compares the full items. This means probing for keys not in the table is cheaper than probing for keys stored in the table. We also compared against F14, the default hash table used at Facebook [Bronson and Shi]. The results are extremely similar and so we include only results with SwissTable.

For **Bloom Filters**, we implemented register blocked Bloom filters from [Lang et al., 2019]. To cut down hashing time, and thus to be conservative with respect to our benefits, we used a variant of double hashing wherein we compute one 64 bit hash function, split it into two 32-bit hash values, and then use these as the inputs to double hashing [Kirsch and Mitzenmacher, 2006]. We also utilize the techniques for fast modulo reduction by multiplication from [Ross, 2007].

For **partitioning**, many of the techniques devised by database research such as software write buffers [Wassenberg and Sanders, 2011] and non-temporal stores [Balkesen et al., 2013] do not apply to large data types or variable length data types. Thus our partitioning is a simple for loop that computes hash values and writes out data directly to a partition.

Base Hash Functions. We use three state-of-the-art hash functions. For hash tables, we use wyhash, which is one of the two default options used in SwissTable. We use both the version contained in SwissTable as well as the most recent optimized version of wyhash given directly by the author [Wang et al., 2020]. For Bloom filters we use xxh3, which is used widely at Facebook and is the default for the Bloom filters in RocksDB [Collet]. For partitioning we use the implementation of CRC32 from the OLAP database Clickhouse [Zheng et al., 2020].

Implementation. We modify each of the three base hash functions. We maintain their basic interface (input is an array of bytes plus a key length), and tightly integrate Entropy-Learned Hashing. Thus there is Entropy-Learned xxh3, Entropy-Learned wyhash, and Entropy-Learned CRC32. The bytes chosen to hash are selected at hash function construction and stored in a const array. The functions read from $data[locations[i]]$ instead of $data[i]$, and we use templates to generate efficient code for partial-key hash functions using 1,2,3,4,.. words. These templates modify the initial function to reduce branching statements because of the known length of the partial-key. All implementation is in C++. All experiments for hash-based tasks are in-memory since hash-based tasks typically run in-memory. For example, a hash table should always fit in memory to get good performance while a Bloom filter will also typically reside in memory to protect from expensive disk access. Thus such structures are both created and utilized in memory. When disk is involved, the CPU cost of hashing is typically not highly visible in terms of operational latency unless very fast

Processor	Intel Xeon E7-4820 v2
#sockets	4
#cores per socket	8
Hyper-threading	2-way
Turbo-boost	Off
Clock speed	2.00GHz
L1I / L1D (per core)	32KB / 32KB
L2 (per core)	256KB
L3 (shared)	16MB
Memory	1TB

Table 3.2: Server Parameters

Dataset name	Avg. key length	# keys
UUID	36	100K
Wikipedia	129	22K
Wiki	22	99K
HN URLs	75	247K
Google URLs	81	1.2M

Table 3.3: Real-world data.

disk devices such as SSDs are used (although CPU usage is still reduced).

Datasets. We use five real-world datasets for experimentation. Two datasets consist of URLs, with one containing the URLs of stored Google Landmarks and the other all URLs posted to Hacker News during 2015 [Noh et al., 2016, Hac, 2015]. The other three, UUID, Wikipedia, and Wiki, are database columns taken from a recent research study [Boncz et al., 2020]. They contain universally unique identifiers, sampled text from Wikipedia, and Wikipedia entry titles respectively. Table 3.3 presents the number of items and average key length for each real-world dataset. In addition, we use synthetic data to have finer control over key size and data size. Section 3.6.3 uses 80 byte keys with bytes 32-39 drawn randomly from the alphabet (26 possible values), and all other bytes constant. Section 3.6.6 uses 8KB keys with each byte ideally random.

Experimental Setup. We use an Intel Ivy Bridge server. Table 3.2 summarizes the server parameters. We use Debian GNU/Linux 10 operating system. Data structures are queried for a warmup phase before timing and input keys for queries are in cache. We pin the thread to a particular core and locally allocate memory. We use Intel VTune's uarch-exploration [Intel, 2021] for performing hardware-level time breakdown and Linux perf [Linux, 2021] for performing memory-level parallelism tests and software-level time breakdown.

3.6.2 Number of Words vs. Entropy

Before demonstrating performance results, we first make the idea of surplus randomness more concrete with examples from real data. We show that for many datasets with medium-sized keys, good hashing properties can be achieved for data structures with millions of elements while hashing only parts of the keys. We divide each dataset in Table 3.3 in half. We use the first part to choose which bytes to hash in a greedy manner as described in Section 3.3. This produces an ordered list of bytes (or words) to choose. Choosing more bytes from the list produces a partial-key function providing more entropy. We use the second half of the dataset to get an unbiased estimate of the entropy for each combination of bytes as described in Section 3.3.

Figure 3.4a shows that the entropy of the result of the partial-key function increases for all datasets with the number of words included. We see that by 3 words being included all datasets have an entropy of at least 18, and 3 of the 5 have entropies above 25. For Wikipedia and UUID, infinite entropy is estimated because no collisions are observed with the partial-key function. Figure 3.4b shows how this entropy translates into data structures, where we see that the Google URLs dataset is capable of using partial-key hashing with hundreds of millions of elements while hashing just a couple words. Similar results can be seen by transposing the other four datasets onto Figure 3.4b, with most datasets supporting hash data structures larger than the actual number of elements found in the dataset.

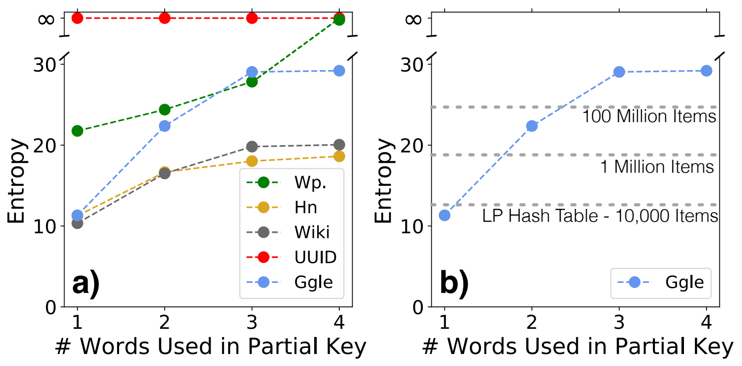


Figure 3.4: The entropy of a dataset grows quickly with the amount of words being hashed. By 4 words, most datasets support data structures with millions of elements.

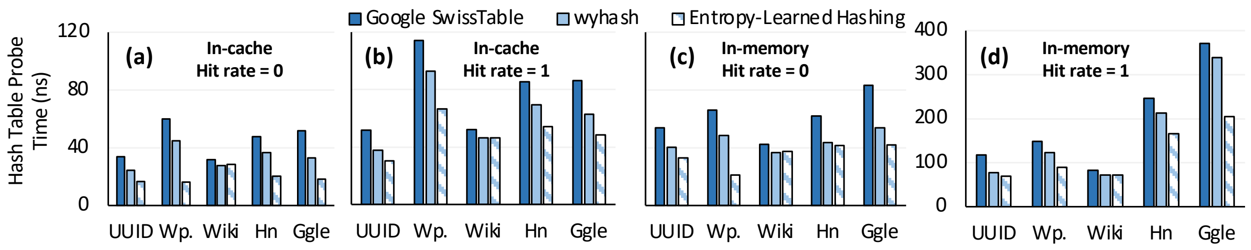


Figure 3.5: Entropy-Learned Hashing reduces probe times for hash tables across datasets, data sizes, and hit rates.

3.6.3 Hash Table Probe Time

After showing that datasets have enough entropy for partial-key hashing to be used,we now turn to showing the performance benefits which can be gained by using Entropy-Learned hash functions for data structures and algorithms. We first focus on hash tables. We examine the probe time per hash table lookup, where we perform the lookups one after the other without any blocking, e.g., similar to the probe-phase of the hash join algorithm.

Entropy-Learned Hashing Reduces Hash Table Probe Time. We first test hash table probe times on real-world datasets for small (L1-resident) and large data (L3/DRAM-resident) with 0% (hit rate = 0) and 100% (hit rate = 1) hit rates. We test with Google's SwissTable using three hash functions: (i) the default hash function provided by SwissTable (GST), (ii) the most recent version of wyhash (FK), and (iii) the Entropy-Learned wyhash hash function (ELH). The small data contains one thousand keys, and the large data contains half of the number of keys of the dataset (we use the other half to generate probes for missing keys). Figure 3.5 shows the results, wherein Entropy-Learned Hashing provides speedups across all data sizes, datasets, and hit rates over full-key hashing. Across the 20 experiments, the average speedup using ELH over wyhash and SwissTable's default hash function is 1.40×, with these speedups being as high 3.7× over the default hash function of SwissTable and as high as 2.9× over wyhash, both of which are well engineered functions and implementations.

Entropy-Learned Hashing Scales with Entropy, not Key Size. To understand the reasons behind the speed up observed in Figure 3.5, we first need to return to Table 3.3 and Figure 3.4. For full-key hashing, it needs to hash each byte of the dataset, and so the number of bytes processed is

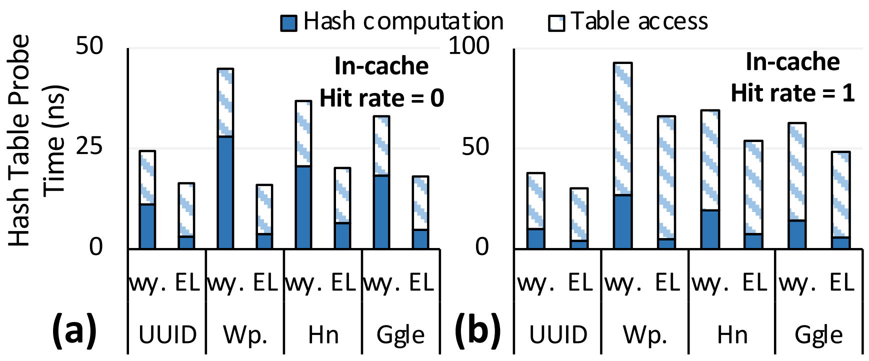


Figure 3.6: Entropy-Learned Hashing significantly reduces computation time bringing speedup as high as 2.9× for cache-resident hash tables with (a) low and (b) high hit rates.

on average the key length given in Table 3.3. For Entropy-Learned Hashing, the number of bytes it hashes is when the entropy of the dataset (seen in Figure 3.4a) crosses the entropy needed by the data structure (seen in Figure 3.4b). When there is a large gap between these two numbers, Entropy-Learned Hashing produces large speedups. For instance, the large gap between the number of bytes hashed is why ELH achieves 2.9× speedup over wyhash and 3.9× speedup over default SwissTable in Figure 3.5a. Similarly, it is why ELH is 1.67× faster than wyhash and 1.81× faster than default SwissTable on the Google dataset in Figure 3.5d.

While faster hashing computation uniformly brings speedups to hash table probes, the amount of this speedup depends on other factors of hash table queries, namely the hit rate and hash table size. We now explain how the combination of these factors with Entropy-Learned Hashing affects performance.

Computation Dominates for Cache-Resident Hash Tables. For cache-resident hash tables, memory requests return quickly and so computation dominates the overall cost of probes. In this case, the savings created by Entropy-Learned Hashing depend on how much work there is beyond the hash function evaluation. Figure 3.6 shows how the work beyond hashing differs for queries for non-existing keys and for existing keys. When queries are for non-existing keys, computation usually consists of the hash function plus small amounts of computation using the tag bits. As Figure 3.6a shows, in this case the hash function evaluation is most of the cost and Entropy-Learned Hashing brings significant benefits. This explains the 1.5×, 2.9×, 1.8×, and 1.8× speedup over wyhash seen in Figure 3.5a for the UUID, Wikipedia, Hacker News, and Google datasets, respectively.

When queries are for keys in the dataset, Figure 3.6b shows the comparison after the hash function evaluation takes significant time. As a result, Entropy-Learned Hashing still provides benefits but not quite as large as before, with the savings being 1.23×, 1.41× 1.28×, and 1.28× for the UUID, Wikipedia, Hacker news, and Google datasets, respectively. Thus in cache, Entropy-Learned Hashing provides up to a 40% speedup for queries on existing keys and up to a 3× improvement on non-existent keys.

Memory Parallelism Dominates for Large Hash Tables. At large data sizes, the increase in computational performance from faster hashing leads to more efficient use of the memory hierarchy. This is due to the effects of CPU pipelining. Namely, when hash table lookups are done one after the other without blocking, then the CPU typically pipelines multiple hash table lookups which are then executed in parallel [Kocberber et al., 2015]. Entropy-Learned Hashing reduces the amount of computation required, and as a result, the CPU fits a larger number of hash table lookups into its pipeline. The effect of this increased pipelining is what creates the speedups seen at large data sizes in Figure 3.5c and 3.5d across datasets, with Entropy-Learned Hashing being as much as 1.67× faster than the nearest competitor.

The amount of this savings depends on the costs of memory accesses, with more expensive memory accesses leading to larger improvements. For instance, in Figure 3.5d we see that the larger datasets Google and Hacker News produce greater savings than the smaller datasets Wikipedia, UUID, and Wiki. Similarly, comparing Figure 3.5d to 3.5c, querying for existing keys produces greater savings because we view both tag bits and full-keys in comparison to just the tag bits most often for missing keys.

Figures 3.7a and 3.7b refine this analysis. Figure 3.7a shows the memory-level parallelism (MLP), which is defined as the number of L1 data cache misses per CPU cycle, for the Hacker News and Google datasets using hit rate = 1. The higher MLP in 3.7a indicates that a large number of data cache misses are being executed in parallel by Entropy-Learned Hashing than by full-key hashing. Figure 3.7b shows how this affects the overall runtime of hash table probes under the same setup, with Entropy-Learned Hashing reducing both the number of instructions executed and memory waiting time. This analysis corroborates the results seen in Figure 3.5c and 3.5d, where

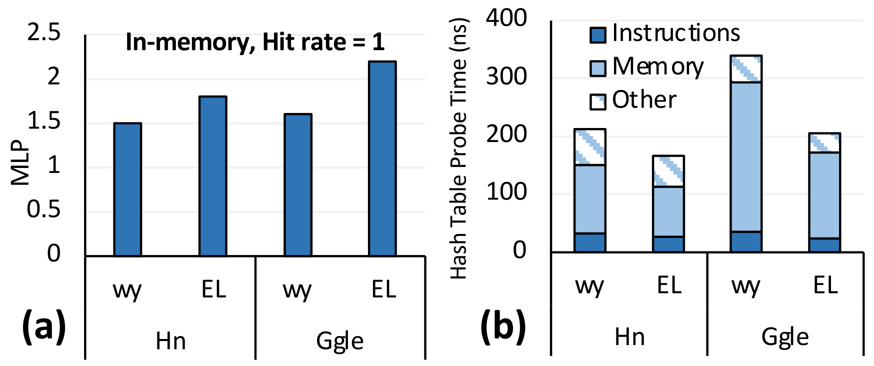


Figure 3.7: (a) MLP is significantly higher for ELH than it is for full-key hashing. (b) As a result, ELH reduces both the number of instructions executed and memory waiting time.

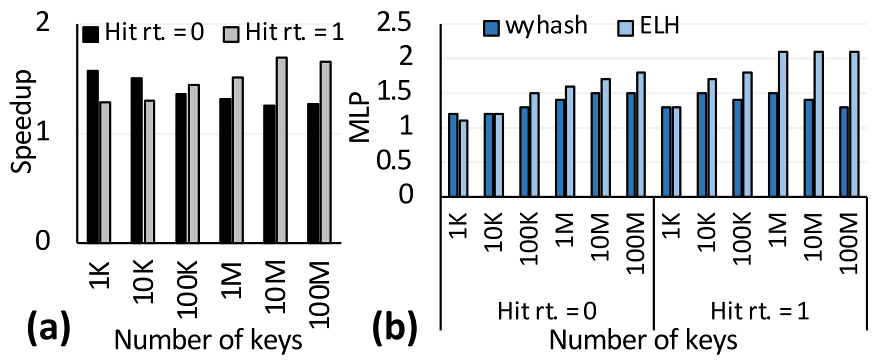


Figure 3.8: (a) Entropy-Learned Hashing provides larger benefits for missing keys at small data sizes and larger benefits for existing keys at large data sizes. (b) Entropy-Learned Hashing improves memory-level parallelism.

Entropy-Learned Hashing provides a 1.31× speedup on average over full-key hashing.

Entropy-Learned Hashing Scales with Data. We now turn to experiments with synthetic data so that we can more finely control the data size and experiment with larger data sizes. Figure 3.8a shows the main result, which is that Entropy-Learned Hashing provides benefits for hash tables across small and large data sizes. At small data sizes of 1K tuples, Entropy-Learned Hashing provides 2.33× speedups on queries for non-existing keys and 1.30× speedups for existing keys. For large data sizes of 100M tuples, this speedup is 1.3× for missing keys and 1.7× for existing keys. Figure 3.8b shows that the reason for these speedups is as discussed before for the real-world datasets. Namely, at small data sizes the savings in computation directly produce speedups for Entropy-Learned Hashing, whereas for large data sizes the more efficient hash computation leads to better MLP which produces faster probe times.

3.6.4 Bloom Filter Lookup Time & FPR

In this section, we evaluate Entropy-Learned Hashing for Bloom filters. We examine the lookup time and false positive rate (FPR) metrics. As input parameters, we let the FPR of the filter be 3% and allow the Entropy-Learned Hashing filter to deviate in FPR by 1%. The filter uses 3 hash functions, but computes only 1 due to double hashing. All parameters are tunable; this experimental setup is meant to reflect high-throughput filters such as those in filter push-down before joins [Lang et al., 2019]. For the small data size we use 1K keys and for the large data size we again use half the number of keys in the data.

Entropy-Learned Hashing Reduces Filter Lookup Time. Figures 3.9a and 3.9b present results for Bloom filters lookup time and FPR using xxHash and Entropy-Learned Hashing. Figure 3.9a shows that Entropy-Learned Hashing improves performance on high entropy datasets such as Google, Hacker News, UUID, and Wikipedia. The speedup is consistently between 1.85× and 4.51×. For Wiki, which has both small key size and low entropy, the speedup is small. Across all datasets, the average speedup is 2.10×, so that Entropy-Learned Hashing consistently provides drasticaly faster throughput on Bloom filter queries.

Entropy-Learned Hashing has Tunable Added FPR. Figure 3.9b presents the FPR of Bloom filters using Entropy-Learned Hashing and full-key hashing. Most importantly, as can be seen in Figure 3.9b, the FPR is within 1% as our tuning parameter suggests so that our analytical bounds hold. Additionally, Figure 3.9b shows that the increase in FPR is usually much less than this tuning parameter, in this case being only 0.1%. Thus, for most datasets the difference in FPR is negligible. Additionally, this FPR increase can be adjusted down or up as needed. Reducing the allowed increase in FPR increases the entropy needed and so requires more hash computation, and so this represents a tunable FPR vs. speed tradeoff.

Bloom Filters require more entropy than Hash Tables. For a dataset size of n and added FPR of ε, ELH requires $\log_2 n + \log_2(1/\varepsilon)$ entropy, which is approximately $\log_2(1/\varepsilon)$ more entropy than hash tables. For certain datasets such as Wiki or Hacker News, this goes beyond the entropy they can provide using small partial-keys and so they revert to using full-key hashing at large data sizes as can be seen in Figures 3.9a and b. For Google URLs, Wikipedia, and UUID, they have

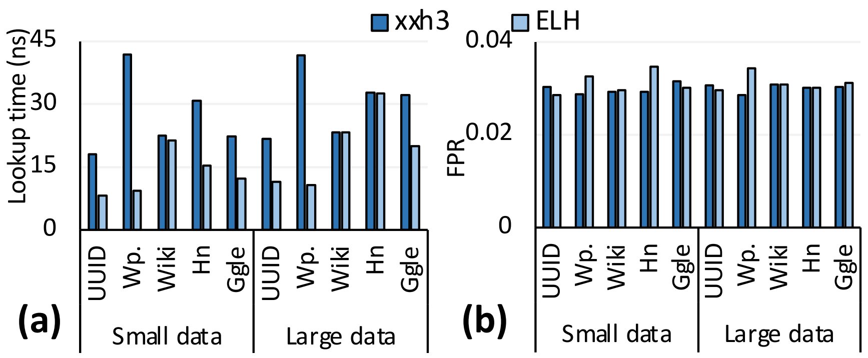


Figure 3.9: Improving Bloom filter lookup time (a) and false positive rates (b) for small and large data sizes.

more than enough entropy and each can support at least 100× more data or a 100× lower added FPR. Thus, these datasets maintain consistent speedups at no cost to FPR for very large data sizes as seen in Figure 3.9b.

3.6.5 Partitioning Time & Variance

Partitioning is used in many contexts. For instance, tuples may be sent across the network in settings such as map-reduce or simply partitioned in memory as in radix-partitioning before hash joins. Because of this, the cost of partitioning depends very heavily on the application it is used in. To help guide users in terms of whether Entropy-Learned Hashing can be useful for their application, we provide three micro-benchmarks. These benchmarks show the increased computational efficiency of Entropy-Learned Hashing on partitioning and put this computational efficiency in context. In the first micro-benchmark, we only compute the partition assigned to each input key. In the second, we keep a list of positional identifiers for each partition and write out the position of each key assigned to each partition. In the third, we write out the actual keys assigned to each partition. As we progress through the microbenchmarks, we move from a computationally heavy task with few writes to a memory bandwidth intensive task which is mostly memory bound. Depending on the setup, the benefit in performance from using Entropy-Learned Hashing may be between 14× and 18%. Thus, the benefit of Entropy-Learned Hashing for partitioning depends on whether the saved computational cycles are of use, either directly through speedups on the task at hand, or indirectly, by allowing other computation to take place while network or memory I/O is being performed.

	Pure hashing		Pos. id.		Data	
# Par.	64	1024	64	1024	64	1024
UUID	3.15	3.15	2.05	1.38	1.01	1.00
Wp.	14.10	14.09	6.18	2.66	1.23	1.18
Wiki	1.25	1.09	1.37	1.10	1.01	1.01
Hn	4.29	1.00	2.72	1.00	1.17	1.03
Ggle	7.83	7.82	2.51	1.42	1.01	1.00

Table 3.4: Speeding up when partitioning.

Like Bloom Filters, partitioning has a tunable parameter which allows the variance (equivalently standard deviation) to increase in exchange for faster hashing. We set this parameters so that each partition is expected to be within 5% of its mean.

Entropy-Learned Hashing Reduces Partitioning Time. Table 3.4 presents the speedups of Entropy-Learned Hashing for the three configurations we examine. Entropy-Learned Hashing dramatically improves the hashing computation as can be seen by the left side of Table 3.4, with increases in speed of above 3× for 4 of the 5 datasets and speedups of up to 14.1×. Partitioning by writing out positional identifiers, seen in the middle column of Table 3.4, is similar, with increases in speed of greater than 2× for 4 of the 5 datasets and speedups of up to 6.2×. Thus, the results show that the computational cost of partitioning is significantly cheaper using Entropy-Learned Hashing. At the same time, writing out large amounts of data can limit the benefits of using ELH for partitioning, as seen in the right side of Table 3.4. By writing out long-key strings at each iteration of the partitioning, limitations on write bandwidth limit gains from Entropy-Learned Hashing. Still, even in this case the speedups can be as much as 20%, and additionally CPU usage is reduced which frees up the CPU for other tasks.

Partitioning quality is maintained using Entropy-Learned Hashing. Table 3.5 presents normalized relative standard deviation for partitioning, where relative standard deviation is obtained by dividing the standard deviation by the average. We calculate relative standard deviation for both full-key and Entropy-Learned Hashing and normalize the Entropy-Learned Hashing to the full-key hashing. As Table 3.5 shows, the normalized relative standard deviations concentrate around one, which shows that the partitions produced by the full-key hashing and the partitions produced by the entropy-learned hashing are similar. In the case they are not, such as for Hacker News with

	Pure hashing		Pos. id.		Data	
# Par.	64	1024	64	1024	64	1024
UUID	1.44	0.95	1.44	0.95	1.44	0.95
Wp.	0.92	1.02	0.92	1.02	0.93	1.02
Wiki	1.35	1.01	1.35	1.01	1.35	1.01
Hn	2.06	1.00	2.06	1.00	2.05	1.00
Ggle	1.09	1.08	1.09	1.08	1.09	1.08

Table 3.5: The relative standard deviations of Entropy-Learned Hashing and full-key hashing are similar.

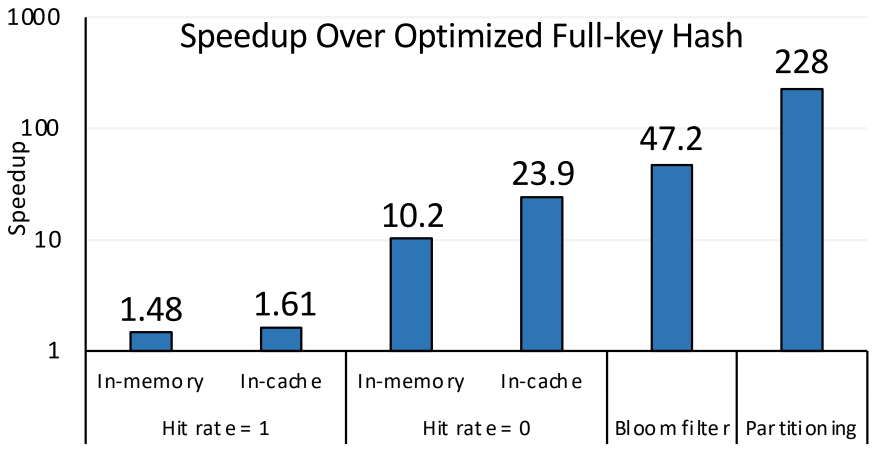


Figure 3.10: Entropy-Learned Hashing provides orders of magnitude speedups with large key sizes.

64 partitions, the relative standard deviation of Entropy-Learned Hashing is less than 3% so that partitions are within 3% of their expected number of items on average.

3.6.6 Large Key Experiments

A key benefit of Entropy-Learned Hashing is that it creates hash functions whose runtime is independent of key size. While this provides computational benefits for data with medium sized keys such as URLs and text, we now show that the speedup is much larger for large keys such as file blocks. To demonstrate this effect, we repeat our experiments for hash tables, bloom filters, and partitioning but with synthetic random keys of 8192 bytes each. Figure 3.10 shows the results. For hash tables with all successful lookups, the benefits of Entropy-Learned Hashing are naturally bounded because the need to compare full keys limits the throughput of this task. However, for hash table lookups that are misses, Bloom filter probes, and partitioning, Entropy-Learned Hashing brings a large speedup that is unbounded and can be one to two orders of magnitude.

3.6.7 Training Time

We now demonstrate that the training time needed for Entropy-Learned Hashing is not a bottleneck. We use the full Google dataset. Table 3.6 shows the results, displaying algorithm run times for a naive implementation which keeps all data points at each iteration, and for our optimized implementation which discards unique keys after each iteration. There are three main takeaways. First, the training time is reasonable for all sizes of contiguous bytes chosen, with runtimes between several minutes and several seconds. Second, pruning items which are unique from the dataset after each iteration produces substantial runtime benefits (if an item is unique on some subset of bytes, adding new bytes cannot create a collision for that item). Third, as the size of the contiguous byte locations we choose increases, the runtime decreases significantly because there are fewer options at each iteration and because after fewer iterations the number of data items that are non-unique is low (making each step, i.e. Algorithm 2, fast).

# bytes	1	4	8
Optimized	214 s	11.6 s	6.4 s
Naive	29 min.	13 min	5 min

Table 3.6: Training runtime

3.7 Conclusion

With the experimental evaluation complete, we recap Entropy-Learned Hashing. Entropy-Learned Hashing started with a simple observation: that if we knew properties of the data, hashing every byte is not necessary. The framework of Cerebral Data Structures then uses this intuition to build out Entropy-Learned Hashing, creating formalisms for how to choose bytes to hash, how the parameter of Rényi entropy affects data structure performance, and how to use these parametric equations plus samples of data to design hash functions. The end result is better hash performance, with this improvement intuitively seen to grow as key sizes grow, and a fundamental breakthrough in hash function evaluation time from being linear in the size of the key to dependent on the entropy in the input bytes.

Chapter 4

Cerebral Filters: Stacked Filters

This chapter introduces our second example of designing cerebral data structures, Stacked Filters, which which looks at how identifying a set of frequently queried negative items in a workload leads to filter data structures that can produce no false positives on this set without adding (substantial) overhead to the space or computation of a filter and without sacrificing robustness. We start by giving an overview of filters and their use cases before showing how the framework of Cerebral Data Structures leads to the design for Stacked Filters.

4.1 Learning to Filter By Structure

4.1.1 The uses of Filter Structures

The storage and retrieval of values in a data set is one of the most fundamental operations in computer science. For large data sets, the raw data is usually stored over a slow medium (e.g., on disk) or distributed across the nodes of a network. Because of this, it is critical for performance to limit accesses to the full data set. That is, applications should be able to avoid accessing slow disk or remote nodes when querying for values that are not present in the data. This is the exact utility of filter structures, also known as approximate membership query (AMQ) structures. Filters have tunably small sizes, so that they fit in memory, and provide probabilistic answers to whether a queried value exists in the data set with no false negatives and a limited number of false positives.

For all filters, the probability of returning a false positive, known as the false positive rate (FPR), and the space used are competing goals. Filters are used in a large number of diverse applications such as web indexing [Goodwin et al., 2017], web caching [Fan et al., 2000], prefix matching [Dharmapurikar et al., 2003], deduping data [Deng and Rafiei, 2006], DNA classification [Stranneheim et al., 2010], detecting flooding attacks [Geneiatakis et al., 2009], cryptocurrency transaction verification [Gervais et al., 2014], distributed joins [Ramesh et al., 2008, Quoc et al., 2018], and LSM-tree based key-value stores [Dayan et al., 2017], amongst many others [Tarkoma et al., 2012].

4.1.2 Prior Approaches in Filter Design

Traditional Filters. As covered in Section 2.3, traditional filter designs (which we will call query-agnostic designs) utilize only the data set during construction. For example, a Bloom Filter [Bloom, 1970] starts with an array of bits set to 0 and using multiple hash functions per value, hashes each value to various positions, setting those bits to 1. A query for a value x probes the filter by hashing x using the same hash functions and returns positive if all positions x hashes to are set to 1. When querying a value that exists in the data set (a positive value), these bits were set to 1 during construction and so there are no false negatives; however, when querying a value not in the data set (a negative value), a false positive can occur if the value is hashed entirely to positions which were set to 1 during construction.

If the hash functions are truly random, then every query-able value not in the data set has the same probability of being a false positive (this is also true of other query-agnostic filters such as Cuckoo [Fan et al., 2014], Quotient [Pandey et al., 2017], and Xor [Graf and Lemire, 2019] filters). This makes query-agnostic filters robust, as they have the same expected performance across all workloads, and easy to deploy, as they require no workload knowledge. However, at the same time, this limits the performance of query-agnostic filters, as they are required to work for any query distribution. Additionally, the possibility for further improvements in the trade-off between space and false positive rate are limited, as current query-agnostic filters are close to their theoretical lower bound in size [Carter et al., 1978, Broder et al., 2002].

Learning-Based Filters. Classifier based filters such as Weighted Bloom Filters [Bruck et al., 2006, Wang et al., 2015], Ada-BF [Dai and Shrivastava, 2020], and Learned Bloom Filters [Kraska et al., 2018, Rae et al., 2019] utilize workload knowledge, making it possible to move beyond the theoretical limits of query-agnostic filters. Such filters need as input a sample of past queries and using that they train a classifier to model how likely every possible value is to 1) be queried, and 2) exist in the data set. The classifier is then used in one of two ways.

In the first [Kraska et al., 2018, Rae et al., 2019], it acts as a module which accepts values that have a high weighted probability of being in the data. It cannot reject values as the stochasticity of the classifier might cause false negatives. Thus, a query-agnostic filter is also built using the (few) values in the data set for which the classifier returns a false negative. Queries are first evaluated by the classifier, and if rejected, then probe the query-agnostic filter. In the second [Bruck et al., 2006, Wang et al., 2015, Dai and Shrivastava, 2020], the classifier uses the weighted probability of being in the set to control the number of hash functions used by a Bloom filter for each value. For values with a high likelihood of being in the set, few hash functions are used, setting fewer bits in the filter but also checking fewer bits, and therefore providing fewer chances to catch a false positive. For values not likely to be in the set, this is reversed.

Classifier based filters can reduce the required memory to achieve a given false positive rate when keys and non-keys have clear semantic differences. URL Blacklisting is such a use case [Kraska et al., 2018], wherein browsers such as Google Chrome maintain a list of dangerous websites and alert users before visiting one. Browsers store a filter containing the dangerous websites at the client. For each web request, the client checks the filter; if the filter rejects the query, the website is safe and can be visited. If the filter accepts the query, an expensive full check to a remote list of dangerous websites is done. Because there are no false negatives, every dangerous website is caught, and that there are only a few false positives means most safe websites do not need to perform extra work.

However, classifier based filters offer a host of new problems. First, the classifier has to be accurate, which can be hard: for some workloads the data value and its probability of existing in the data set are loosely correlated. For other types of data that appear in practice such as hashed IDs, there is no correlation. Additionally, even when data patterns exist, often data such as textual keys have complex decision boundaries which require complex models such as neural networks for accurate classification. As a result, the computational expense of the classifier is often orders of magnitude more expensive than hashing. Finally, the classifier is trained on a specific sample workload, and thus if the workload shifts, we need to go through the expensive process of gathering sample queries and retraining the classifier to maintain good performance.

4.1.3 Cerebral Filters: Stacked Filter Structures

We now use the framework of Cerebral Data Structures to design Stacked Filters, a novel filter data structure. We start by reasoning through how the parameterization of prior approaches affects their designs, and use this reasoning to design a new parameterization of the workload that leads to superior overall performance. In particular, by learning about the workload it maintains the superior tradeoffs between FPR and space of learned filters, while the parameterization allows for the removal of the classifier from the end structure, preserving the computational benefits and robustness of traditional filters.

To get to our end design, we start by analyzing the prior approaches. For traditional filters, given that they have only knowledge of the data in the set being stored, they have a clear place to improve: the false positive rate (step 1). This is because they lack any parameterization of the workload, and so cannot tailor the data structure to make more frequently queried items less likely to be false positives (step 2a). In contrast, Learned filters have great false positive rate vs. size tradeoffs, but have issues with computational speed and robustness (step 1). This is because they have a very rich parameterized model of the workload, requiring as part of data structure operations an estimate of the probability for an item to be in the filter. This in turn requires a classifier to to be queried as part of data structure operations, causing the aforementioned problems with speed and robustness (step 2a).

To enable the benefits of learning-based approaches without the classifier, Stacked Filters reduces the parameterization of the workload to a simpler representation: that of a single set of frequently queried negative values and how likely this set is to be queried (step 2b). Then, using this simpler representation, we build a novel data structure consisting of a sequence of filters which alternate between representing values from the positive set and from the set of frequently queried negative items. The structure makes it so that all frequently queried negative need to go through multiple membership checks to be false positives, exponentially reducing their probability of being a false positive. At the same time, each layer in the sequence of filters is both exponentially smaller in size and exponentially less likely to be queried, and so both size and computation costs rise like geometric series. As a result, these costs are close that of a single filter (step2b). The overall result is that for workloads with any frequently queried non-existing values, Stacked Filters provide a superior tradeoff between false positive rate, filter size, and computation than either of classifier-based filters or query-agnostic filters. In particular, they match the false positive rate vs. size tradeoffs of learned filters while being just as computationally efficient and robust as traditional filters.

The rest of this chapter builds on the idea described above. In particular, it explains in more detail how this stack of alternating filters works, and details the performance metrics of the overall Stacked Filter structure given the parameters of the workload (step 2b). It also explains how to estimate query probabilities from a sample of past data in order to build this set of frequently queried negatives (step 3), how to choose what negative items should be in this set, and how to optimize the overall Stacked Filter structure given the parameters of the workload (step 4). The result is that given the implementation of each of these steps, we have an overall process which takes in samples of past queries and produces and optimized filter structure for the task at hand.

Contributions. The major contributions of the Cerebral Data Structures approach to filters are:

- *Data Structure Formalization*: We introduce a new way to design workload-aware filters as multi-layer filter structures which index both positives and frequent negatives.
- *Generalization*: We show that Stacked Filters work for all query-agnostic filters including Bloom, Cuckoo, and Quotient Filters.
- *Better trade-off of FPR and size*: We derive the metric equations of Stacked Filters for size, computation, and false positive rate. Using these equations, we provide theoretical results showing Stacked Filters are strictly better in terms of FPR vs. size than query-agnostic filters on the majority of workloads, and quantify the expected benefit.
- *Optimization*: We show that the optimization problem of tuning the number of layers and

layer sizes is non-convex. Still, we provide ϵ-approximation algorithms, running in the order of milliseconds, which automatically tune the number of layers and the individual sizes of each layer so that performance is arbitrarily close to optimal.

- *Adaptivity*: We show that the benefits of Stacked Filters can be extended to Stacked Filters built adaptively. Here, Stacked Filters start with a rough knowledge of how skewed a workload is, but not which values are frequently queried, and build their structure incrementally during normal query execution.
- *Experiments*: Using URL blacklisting, a networking benchmark, and synthetic experiments we show that Stacked Filters 1) provide improvements in FPR of up to $100\times$ over the best alternative query-agnostic or classifier-based filter for the same memory budget, while retaining good robustness properties, 2) provide a superior tradeoff between false positive rate, size, and computation than all other filters, resulting in up to $130\times$ better end-to-end query throughput than the best alternative, and 3) for scenarios where learning is not easy, Stacked Filters can still utilize workload knowledge and offer throughput up to $1000\times$ better than classifier-based filters.

4.2 Notation and Metrics

We first introduce notation used throughout the paper and metrics that are critical for describing the behavior of filters. Table 4.1 lists the key variables and metrics.

Notation. Let U be the universe of possible data values, such as the domain of strings or integers. Let P be a data set, which we will refer to as the positive set, and let $N = U - P$ be the set of negative values. From now on we will refer to data values as well as to queried values as elements, which is the traditional terminology in the filters literature. We will denote filter structures by F, which we treat as a function from $U \to \{0, 1\}$, and we say that x is accepted by F if $F(x) = 1$ and that x is rejected by F if $F(x) = 0$.

As a filter, we have $F(x) = 1 : \forall x \in P$ and we are interested in minimizing the number of false positives, which are the event $F(x) = 1, x \notin P$. The filter F is itself random; different instantiations of F produce different data structures, either because the hash functions used have randomly chosen parameters or because the machine learning model used in classifier based filters is stochastic.

Expected False Positive Bound (EFPB). A traditional guarantee for a filter F is to bound $E_F[\mathbb{P}(F(x) = 1 | x \notin P)]$ for any x chosen independently of the creation of F. We call this bound the *expected false positive bound.*

Expected False Positive Rate (EFPR). Given a distribution D over U which captures the query probabilities for elements in U, the *expected false positive rate* is $E_{x \sim D}[E_{F|D}[\mathbb{P}(F(x) = 1 | x \notin P)]]$.

Optimization via Expected False Positive Rate. Query throughput most directly relies on the EFPR and since the EFPR depends on the query distribution D, the goal of workload-aware filters is to capture and utilize D to improve the EFPR. Namely, for $x \in N$ with higher chance of being queried, workload-aware filters should lower the probability that x is a false positive.

Robustness via Bounding False Positive Probability. Optimizing the EFPR helps system throughput but brings concerns about workload shift. Query-agnostic filters can act as a safeguard against such a shift. To see this, note that F and D are independent by assumption and so for a query-agnostic filter with FPR ϵ,

$$E_{x \sim D}[E_{F|D}[\mathbb{P}(F(x) = 1 | x \notin P)]] \leq E_{x \sim D}[E_{F|D}[\epsilon]] = \epsilon$$

regardless of what D is. Thus, filters which provide an expected false positive bound provide an upper bound on the expected false positive rate for any workload D chosen independently of the filter.

Memory - False Positive Tradeoff. For all filter structures, their EFPR and EFPB can be made arbitrarily close to 0 with enough memory, and there exists a tradeoff between the memory required and the false positive rate provided. Thus for purposes of comparison, we always report EFPR and EFPB with respect to a space budget. Space budgets in practice tend to be between 6 and 14 bits per element, and are significantly smaller than the elements they represent (which can be anywhere from 4 bytes to several megabytes).

For query-agnostic filters, the EFPR and the EFPB are equal, and is just called the false positive rate. Additionally, for all query-agnostic filters, the false positive rate α and size in bits per element s are 1-1 functions of each other. When going from one to the other, we denote the quantities by

Notation	Definition
P	Set of all positive elements
N	Set of all negative elements
N_f	Negatives used to construct a Stacked Filter
N_i	The complement of N_f, i.e. $N \setminus N_f$
s	Size of a filter in bits/element
$c_{i,S}$	Cost of inserting an element from set S
$c_{q,S}$	Cost of querying a filter for an element in S

Metric	Definition
EFPR	Expected false positive rate of a filter structure given a specific query distribution
EFPB	Upper bound on the EFPR of a filter structure for queries chosen independently of the filter

Table 4.1: Notation used throughout the paper

$s(\alpha)$ and $\alpha(s)$, which denote the size for a given FPR and the FPR for a given size respectively.

Computational Performance. Filter structures desire computational performance much faster than the cost to access the data they protect. We denote the cost to insert into a filter an element of set S by $c_{i,S}$. We also denote the cost to query for an element of set S by $c_{q,S}$. If no set is denoted, then $S = U$.

4.3 Stacked Filters

We start by describing the structure of stacked alternating filters in this Section. Section 4.4 then provides the equations for this structure given the parameters of the workload. Finally, Section 4.5 shows how to estimate these parameters, how to choose the set of frequently queried negative values, and how to tune the actual structure of stacked alternating filters, closing the loop from workload sample to optimized data structure.

A Different view on Filters. The traditional view of filters is that they are built on a set S, and return no false negatives for S. An alternative view of a filter is that it returns that an element is certainly in $\overline{S}$ (the complement of S), or that an element's set membership is unknown. In Stacked Filters, we use this way of thinking about filters, with S for different layers of the stack alternating between subsets of P and subsets of N, to iteratively prune the set of elements in U whose set membership is undecided.

Stacked Filters by Example. We start with an example of a 3 layer Stacked Filter using Figure

4.1. The filter is given the data set P and a set of frequently queried negatives N_f. The first filter in the stack, L_1, is constructed using P similarly to a traditional filter except with fewer bits per element so as to reserve space for subsequent layers. Conceptually, L_1 partitions the universe U. Items that L_1 rejects are known to be in N and can be rejected by the Stacked Filter. Items accepted by L_1 can have set membership of P or N and thus their status is unknown. If the Stacked Filter ended here after a single filter, as is the case for all query-agnostic filters, all undecided elements would be accepted by the Stacked Filter.

Instead, Stacked Filters construction continues by probing L_1 for each element in N_f. Using all elements of N_f accepted by L_1, and which therefore normally would become false positives, Stacked Filters build a second layer with another query-agnostic filter. During a query, values which are still undecided after L_1 are passed to L_2. If L_2 rejects the value, the value is definitely in $\overline{N_f}$, which includes both P and $N \setminus N_f$, which we denote by N_i and call the infrequently queried negative set. Since $P \cup N_i$ contains both positives and negatives, the overall Stacked Filter accepts all the rejected elements of L_2 in order to maintain a zero false negative rate. If the element is instead accepted by L_2, then its set membership is still undecided and so it continues down the stack.

Construction then continues by querying L_2 for all elements in P and building a third layer with a query-agnostic filter. This layer uses as input all elements of P whose set membership is undecided after querying L_2. At query time, L_3 performs the same operations as L_1; elements rejected by L_3 are certainly in $\overline{P} = N$ and so are rejected by the Stacked Filter.

Workload-aware Design. L_2 and L_3 are how Stacked Filters structurally incorporate workload knowledge. They collaborate to filter out frequent negatives to minimize FPR. All frequent negatives that are false positives on L_1 reach L_3 since they are in the construction set for L_2, and so such frequent negatives need to pass an extra membership check to be false positives for the full Stacked Filter. To make deeper Stacked Filters, and thus perform more checks on frequently queried negatives, we recursively perform this process, adding more paired sets of layers.

An intuitive understanding of the effectiveness of Stacked Filters comes from the interplay between the extra size of additional layers vs. their benefit for FPR. Compare the simple 3 layer example above, assuming each layer has an FPR of 0.01, with a single traditional filter using FPR

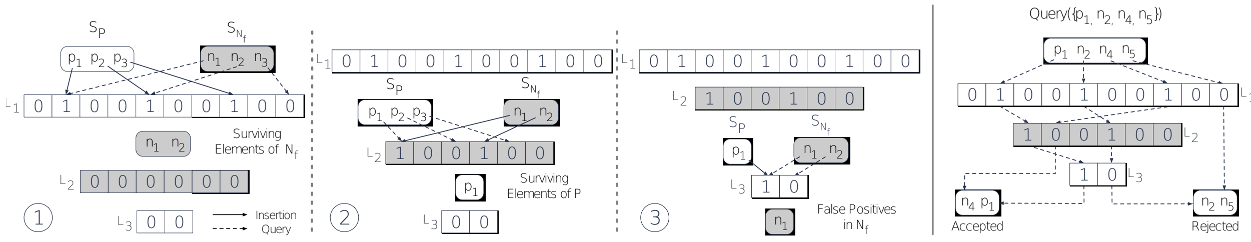


Figure 4.1: Stacked Filters are built in layer order, with each layer containing either positives or negatives. The set of elements to be encoded at each layer decreases as construction progresses down the stack. For queries, the layers are queried in order and the element is accepted or rejected based on whether the first layer to return not present is negative or positive.

0.01. If $|N_f| = |P|$, then L_2 and L_3 are on average 1/100 the size of L_1, and the Stacked Filter has 2% higher space costs than a traditional filter. But for every element in N_f, the extra membership check makes their FPR a full 100× lower; thus if N_f contains any significant portion of the query distribution, the Stacked Filter has a much lower EFPR.

General Stacked Filters Construction. Algorithm 3 shows the full construction algorithm. Like both query-agnostic and classifier based filters, Stacked Filters need two inputs 1) the data set, and 2) a constraint (memory budget or a desired maximum EFPR).

Stacked Filters also need workload knowledge similarly to classifier-based filters. Later on we describe in detail how much knowledge is needed, and how to gather it (even adaptively). For now, we denote workload knowledge abstractly as W, and feed this into an optimization algorithm which returns 1) a set of frequently queried negatives, denoted by N_f, and 2) the number of layers T_L and false positive rate of each layer $\alpha_1, \ldots, \alpha_{T_L}$.

Querying a Stacked Filter. Algorithm 4 formalizes the description given in the example for querying a Stacked Filter. Querying for an element x starts with the first layer and goes through the layers in ascending order. At every layer, if the element is accepted by the layer, it continues to the next layer. If an element is rejected by a layer containing positive elements (called a positive layer), it is rejected by the Stacked Filter. Conversely, if an element is rejected by a layer containing negative elements (negative layer), it is accepted by the Stacked Filter. If the element reaches the end of the stack, i.e. it was accepted by every layer, then the Stacked Filter accepts the element. We now show this algorithm is correct, and explain how the algorithm differently affects positives, frequently queried negatives, and infrequently queried negatives.

Algorithm 3 ConstructStackedFilter(S_P, W, C)

```
Input: $S_P, W, C$: data set, workload, constraint
 1: $S_{N_f}, \{\alpha_i\}$ = $OptimizeSF$(W,C)                              In Sec. 5
 2: // Construct the layers in the filter sequentially.
 3: for $i = 1$ to $T_L$ do
 4:     $S_r = \{\}$
 5:     if $i$ mod $2 = 1$ then                                             layer positive
 6:         construct $L_i$ w/ space $s(\alpha_i) * |S_p|$
 7:         for $x_p \in S_P$ do
 8:             $L_i$.INSERT($x_p$)
 9:         for $x_n \in S_{N_f}$ do
10:             if $L_i(x_n) = 1$ then
11:                 $S_r = S_r \cup \{x_n\}$
12:         $S_{N_f} = S_r$
13:     else                                                                layer negative
14:         allocate $L_i$ w/ space $s(\alpha_i) * |S_{N_f}|$
15:         for $x_n \in S_{N_f}$ do
16:             $L_i$.INSERT($x_n$)
17:         for $x_p \in S_p$ do
18:             if $L_i(x_p) = 1$ then
19:                 $S_r = S_r \cup \{x_p\}$
20:         $S_P = S_r$
21: return F
```

Algorithm 4 Query(x)

```
Input: $x$: the element being queried
 1: // Iterate through the layers until one rejects $x$.
 2: for $i = 1$ to $T_L$ do
 3:     if $L_i(x) = 0$ then
 4:         if $i$ mod $2 = 1$ then
 5:             return reject                          Layer positive, reject x
 6:         else
 7:             return accept                          Layer negative, accept x
 8: return accept                                      No layer rejected, accept $x$
```

Querying a Positive. Stacked Filters maintain the crucial property of having no false negatives. During construction, every positive element x_p is added into each positive layer until it either hits the end of the stack, or is rejected by a negative layer. Querying the Stacked Filter for x_p follows the same path. Either x_p makes it to the end of the stack and is accepted by the Stacked Filter, or it is rejected by some negative layer and thus accepted by the Stacked Filter. Figure 4.1 shows how this process works for the positive element p_1.

Querying a Negative. For an infrequently queried negative element $x_i \in N \setminus N_f$, it is in neither P nor N_f and so has high likelihood of being rejected by both positive and negative layers. As a result, the majority of infrequently queried negatives are rejected at L_1, and the majority of false positives occur from elements rejected at L_2. Frequently queried negatives are a false positive for the Stacked Filter only if they are a false positive for each positive layer. This drives the FPR of frequently queried negatives towards 0, as their false positive rate decreases exponentially in the number of layers. Figure 4.1 shows how this process works for infrequently queried negatives n_4 and n_5 as well as frequently queried negative n_2.

Item Insertion and Deletion. Stacked Filters retain the supported operations of the query-agnostic filters they use. If the underlying query-agnostic filter supports insertion, then so does the Stacked Filter. The same is true for deletion of positives.

Insertion of an element after construction follows the same path as an insertion during the original construction of the filter. The positive element alternates between inserting itself into every positive filter, and checking itself against every negative filter, stopping at the first negative filter which rejects the element. This process works even in the case that the element was previously a frequently queried negative. The deletion algorithm follows the same pattern as the insertion algorithm, except it deletes instead of inserts the element at every positive layer.

4.4 Metric Equations

We now present the metric equations and derivations for Stacked Filters. These equations are then used in Section 4.5 to show how to tune Stacked Filters for optimal performance, and in Section 4.7 to show how Stacked Filters outperform state-of-the-art filters. We show that compared to query-agnostic filters, Stacked Filters are as robust, while improving drastically in EFPR and size.

Notation. Stacked Filters metrics are written as a function of $\vec{\alpha} = (\alpha_1, \ldots, \alpha_{T_L})$ plus an additional dummy $\alpha_0 = 1$ which is used for readability. To distinguish them from the metrics for the base filter, metrics for Stacked Filters are denoted with a prime at the end, so for instance, the size and EFPR of a Stacked Filter are $s'(\vec{\alpha})$ and $EFPR'(\vec{\alpha})$. The metrics are variations on either an exponential function or geometric series. To make this clear by immediate inspection, we will often consider that all α_i values have the same value α (this also makes α and T_L easier to optimize in Section 4.5).

4.4.1 Stacked Filters EFPR

To calculate the total EFPR for a Stacked Filter, we introduce a new variable ψ which captures the probability that a negative element from query distribution D is in N_f, i.e. $\psi = \mathbb{P}(x \in N_f | x \in N)$.

Frequently Queried Negatives. For $x \in N_f$, a Stacked Filter returns 1 if and only if it makes it to the end of the stack. This occurs only if it is a false positive on each positive layer and so the probability of this happening is

$$\mathbb{P}(F(x) = 1 | x \in N_f) = \prod_{i=0}^{(T_L-1)/2} \alpha_{2i+1}$$

Infrequently Queried Negatives. For $x \in N_i$, its total false positive probability is the sum of the probability that it is rejected by each negative layer, plus the probability it makes it through the entire stack. For negative layer $2i$, the probability of rejecting this element is $\prod_{j=1}^{2i-1} \alpha_j \cdot (1 - \alpha_{2i})$, where the first factor is the probability of making it to layer $2i$ and the second factor is the probability that this layer rejects x. Summing up these terms and adding in the probability of making it through the full stack, we have

$$\mathbb{P}(F(x) = 1 | x \in N_i) = \prod_{i=1}^{T_L} \alpha_i + \sum_{i=1}^{(T_L-1)/2} (\prod_{j=1}^{2i-1} \alpha_j)(1 - \alpha_{2i})$$

Expected False Positive Rate. Since N_i and N_f partition N, the EFPR of a Stacked Filter is

$$\psi \prod_{i=0}^{(T_L-1)/2} \alpha_{2i+1} + (1-\psi)(\prod_{i=1}^{T_L} \alpha_i + \sum_{i=1}^{(T_L-1)/2} (\prod_{j=1}^{2i-1} \alpha_j)(1 - \alpha_{2i}))$$

If all α values are equal, then this is equal to

$$EFPR = \psi \alpha^{\frac{T_L+1}{2}} + (1-\psi) \frac{\alpha + \alpha^{T_L+1}}{1+\alpha} \tag{4.1}$$

Thus, the FPR for frequently queried negatives is exponential in the number of layers and goes quickly to 0, whereas infrequently queried negatives have EFPR close to the FPR of the first layer.

4.4.2 Stacked Filter Sizes

Size of a Stacked Filter Given the FPR at Each Layer. For every positive layer after the first, an element from P is added to the layer if it appears as a false positive in every previous negative layer. Thus, the size of all positive layers in bits per positive element is

$$\sum_{i=0}^{(T_L-1)/2} s(\alpha_{2i+1}) \cdot (\prod_{j=0}^{i} \alpha_{2j})$$

Similarly, negatives appear in a negative layer if they are false positives for every prior positive layer and so the size of all negative layers (using the traditional metric bits per positive element) is

$$\sum_{i=1}^{(T_L-1)/2} s(\alpha_{2i}) \cdot \frac{|N_f|}{|P|} \cdot (\prod_{j=1}^{i} \alpha_{2j-1})$$

The total space for a Stacked Filter is the sum of these two equations. Because α_i values are small, the products in parenthesis go to 0 quickly in both equations and so the total size of a Stacked Filter is dominated by its first layer.

Size When Each Layer has Equal FPR. In the case that all α values are the same, we can use a geometric series bound on both arguments above, giving

$$s'(\vec{\alpha}) \leq s(\alpha) \cdot (\frac{1}{1-\alpha} + \frac{|N_f|}{|P|}\frac{\alpha}{1-\alpha}) \tag{4.2}$$

where $s'(\alpha)$ represents the size in bits per (positive) element.

Stochasticity of Size or Filter Behavior. When constructing Stacked Filters, there are two choices when it comes to space. First, all filters can have their memory allocated up front. Using this method, size is fixed but if a higher proportion of elements makes it through the initial layers of the stack, bad behavior can happen at the subsequent filters in the stack. This happens in the form of increased FPR (Bloom filters), failed construction (Cuckoo filters), or long probe times (Quotient filters). Instead, our default is to allocate size proportional to the number of items which make it to a layer in the stack (see lines 6 and 14 of Algorithm 3). This makes the size of a Stacked Filter random, however, for large sets the size of a Stacked Filter concentrates sharply around its mean. In particular,

$$\mathbb{P}(|s' - E[s']| \geq kE[s']) \leq$$
$$\frac{1}{|P|} \cdot \frac{\alpha_{max}}{k^2} \cdot (\frac{s(\alpha_{min})(1-\alpha_{min})}{s(\alpha_{max})(1-\alpha_{max})})^2 \cdot \frac{(1+\frac{|N_f|}{|P|})}{(1+\frac{|N_f|}{|P|}\alpha_{min})^2}$$

where α_{min}, α_{max} are the lowest, highest FPRs of any layer in the Stacked Filter. The proof can be found in Appendix Section B.3.1. The leading $\frac{1}{|P|}$ term ensures that for large sets the chance of deviating away from the expected set size is negligible.

4.4.3 Stacked Filter Robustness

The First Layer Provides Robustness. Any element in N is either in N_i or N_f, and so its probability of being a false positive is either $\mathbb{P}(F(x) = 1|x \in N_i)$ or $\mathbb{P}(F(x) = 1|x \in N_f)$. Since elements of N_i have a higher chance of being a false positive, the EFPB of a Stacked Filter is $\mathbb{P}(F(x) = 1|x \in N_i)$. For a Stacked Filter, an easy bound on this is the FPR of the first layer. Since the majority of the size of a Stacked Filter is in its first layer, worst case performance is similar to a query-agnostic filter (of the same size).

Performance Change Under Workload Shift. While EFPB provides worst case bounds, the EFPR equation shows what happens under the common case of more mild workload drifts. For an initial query distribution D with corresponding ψ, which changes to D' and corresponding ψ', the change in EFPR from D to D' depends only on the change in ψ to ψ'. In particular, the change in EFPR is

$$(\psi' - \psi) \cdot \left(\mathbb{P}(F(x) = 1|x \in N_f) - \mathbb{P}(F(x) = 1|x \in N_i)\right)$$

Thus, the performance in terms of EFPR for a Stacked Filter decreases linearly with the change in the proportion of queries aimed at frequently queried negatives.

4.4.4 Stacked Filter Computational Costs

Like the previous derivations, the resulting equations for query computation time and construction time are modified geometric series. We give here bounds on the resulting equations specifically for all α_i equal to α. The derivations for exact equations with general α_i and an arbitrary number of layers are in Appendix Section B.2. All equations are in terms of average computational cost and given as the number of base filter operations required.

Construction. The construction cost of Stacked Filters is

$$|P|(c_i + c_q)\frac{1}{1-\alpha} + |N_f|c_q + |N_f|(c_i + c_q)\frac{\alpha}{1-\alpha}$$

Like in the previous subsections, filters after the first add only negligible costs to construction to Stacked Filters when α is small. The majority of the cost above comes from 1) $c_i \cdot |P|$ for constructing the first layer and 2) $c_q \cdot (|N_f| + |P|)$ for querying the first layer.

Query Costs. The costs for querying a Stacked Filter for a positive, frequently queried negative, and infrequently queried negative are:

$$c'_{q,P} \leq \frac{2}{1-\alpha}c_q, \quad c_{q,N_f} \leq \frac{1+2\alpha}{1-\alpha}c_q, \quad c'_{q,N_i} \leq \frac{1}{1-\alpha}c_q \tag{4.3}$$

For small α, the cost of querying negative elements is essentially identical to querying a single filter. The cost of querying positive elements is about 2× the cost of a single filter as they make it through the first layer with certainty before being rejected at layer 2 with high probability.

4.5 Optimizing Stacked Filters

Sections 4.3 and 4.4 introduced Stacked Filters given the parameters of the workload: the set of frequent negatives and how likely it is to be queried, and the parameters of the structure: the FPRs of each layer, and the number of layers. We now complete the picture of how Stacked Filters are constructed. This is done in two stages: first we go from a sample of past queries to a workload model, and then we go from a model of the workload to the choice of Stacked Filters parameters.

4.5.1 Modeling the Workload

Like classifier-based filters, Stacked Filters require workload knowledge in the form of a sample of past queries. This set of past queries can have multiple sources depending on the application. For instance, it can be: 1) publicly available, as in the case of URL blacklisting [Kraska et al., 2018] with popular non-spam websites and their query frequencies collected by OpenPageRank [dom], 2) collected by the application by default, as is the case for web indexing and document search [Goodwin et al., 2017], where query term frequencies are collected and stored, or 3) can be collected by the system by choice, as is the case for most data systems including key-value stores [RocksDB, 2020].

After collecting the set of sample queries, Stacked Filters create a model to identify frequently queried elements. They do this by creating a smoothed histogram of the empirical query frequencies. More specifically, each element observed in the set of sample queries is put into a set N_{samp}. Then, the proportion of queries at elements outside N_{samp} is estimated by looping over all possible subsets of $Q-1$ queries from the set of Q sample queries, and seeing for what proportion of subsets the Qth query value is not present in the set of $Q-1$ queries. If we denote this value by ℓ, then for each $x_i \in N_{samp}$, its query frequency is estimated as $\frac{(1-\ell)}{Q} \cdot \sum_{j=1}^{Q} \mathbb{1}_{q_j = x_i}$. Our optimization algorithms below then choose some of the values in N_{samp} to be in N_f, creating the frequent negatives set.

4.5.2 Optimization Algorithms

The final step in constructing Stacked Filters is to use the workload model to choose N_f, T_L, and $\{\alpha_i\}_{i=1}^{T_L}$. The optimization algorithms which do so depend on the base filter being used and fall into two categories. In the first, the query-agnostic filter can take on any value of α, which is a good approximation for filters such as Bloom Filters. In the second, the possible α values are of the form 2^{-k} for $k \in \mathbb{N}$, which is true or a very close approximation for fingerprint based filters such as Cuckoo and Quotient Filters. For both methods, we assume that the base filter has a size equation of the form $s(\alpha) = \frac{-\log_2(\alpha)+c}{f}$ with $c \geq 0, f \geq 1$. This holds true or is a very close approximation for all major filters in practice including Bloom, Cuckoo, and Quotient filters, and additionally covers the equation for the theoretical lower bound on size for query-agnostic filters. Throughout the section, optimization is given in terms of minimizing EFPR with respect to a constraint on size. Optimization of size with respect to a bound on EFPR is similar. Additional constraints on the EFPB or expected number of filter checks may be added by only minor modifications.

4.5.2.1 Outer Loop: Sweeping over N_f

For both continuous and discrete FPR filters, there is an outer loop which chooses sets of N_f to optimize and an inner optimization which optimizes the Stacked Filter given N_f. The best performing value of N_f is then used.

To choose N_f, we make use of the workload model and choose N_f to be a subset of N_{samp}. The $\psi(N_f)$ value is the sum of the estimated query frequencies of each value chosen to be in N_f. Because EFPR is a monotonically decreasing function of ψ, for a fixed size N_f it is optimal to greedily choose the negative elements queried most. Thus we can order N_{samp} by the element's query frequencies and then sweep over various sizes for N_f, always using the most frequently queried elements of N_{samp} to be in N_f. The following theorem shows we can choose the size of N_f efficiently. Its proof, and the proof of all other theorems in this Section, is given in the Appendix .

Theorem 4.5.1. *Given an oracle returning the optimal EFPR for a given set N_f, finding the optimal EFPR across all values of $|N_f|$ to within ϵ requires $O(\frac{1}{\epsilon})$ calls to the oracle.*

The core idea of the theorem is that values of $|N_f|$ that are close together have solutions with optimal EFPR close to each other. Using the theorem, our algorithm starts with a "current" N_f of size 0. It then increases $|N_f|$ to a strategically chosen larger value, making sure the skipped values of $|N_f|$ could have EFPR no more than ϵ lower than the checked values, and runs the optimization with the new fixed ψ and $|N_f|$. It continues to do so until $|N_f| = |N_{samp}|$, and returns the setup giving the best observed EFPR.

4.5.2.2 Inner Optimization: Continuous FPR Filters

The inner optimization loop for continuous filters has N_f and ψ given and works in two steps. First, we assume that all layers have the same FPR and optimize the filter as if it had infinitely many layers. Second, we truncate the infinite layer Stacked Filter to a small finite layered one that is close in performance to the infinite layer one. An optional third step modifies the procedure to search filters with varying α values across layers, but we note that this procedure is optional as it generally does not improve the EFPR.

Step 1: Fixed N_f, Infinite Equal FPR Layers. Take the equations of Section 4.4 with FPR equal across layers and let $T_L \to \infty$. The equations for EFPR, size, and EFPB converge to $s'(\alpha) = s(\alpha) \cdot (\frac{1}{1-\alpha} + \frac{|N_f|}{|P|}\frac{\alpha}{1-\alpha})$, $EFPR(\alpha) = (1-\psi)\frac{\alpha}{1+\alpha}$, and $EFPB = \frac{\alpha}{1-\alpha}$, and the equations for computation converge to Equation (4.3).

By inspection, the equations for EFPR, EFPB, and computation monotonically increase with α.

Thus attempts to minimize the EFPR or satisfy EFPB or computation constraints should all lower α. For the size equation, the following theorem holds.

Theorem 4.5.2. *The function* $s'(\alpha) = \frac{-\log_2(\alpha)+c}{f} \cdot (\frac{1}{1-\alpha} + \frac{|N_f|}{|P|}\frac{\alpha}{1-\alpha})$ *is quasiconvex on (0,1) when* $c \geq 0, f > 0$.

Quasi-convex functions have unique global minima, and as a result, the size equation can be minimized via gradient descent. Specifically, we use gradient descent with backtracking line-search to choose the step size. To minimize EFPR using size as a constraint, at a given time step if we are below the size constraint we decrease α. Otherwise, we use the gradient of size with respect to α to decrease the size. If for the given N_f one or more constraints is not satisfiable, we return an exception.

Step 2: Truncating to a Finite Stack. When performing truncation, we measure the difference in each metric equation using infinite T_L and an increasing finite value of T_L and stop when the difference is below ϵ for all metric equations. Because each metric equation is either an exponentially decreasing function of T_L or a geometric series in T_L, the convergence to the infinite layer values is on the order of $O(\alpha^{T_L/2})$, and so usually 5 or 7 layers suffice.

Algorithm Analysis. By using $\frac{\epsilon}{3}$ in both the outer loop over N_f and both steps 1 and 2, the overall algorithm is an ϵ approximation to the best possible EFPR for a Stacked Filter with α fixed across layers. Its runtime is $O(\epsilon^{-1} + |N_{samp}|)$ and its empirical optimization times for Stacked Bloom Filters at 10 bits per element are listed in Table 4.2 under "Bloom, fixed α". The workload, described in detail in Section 4.8.2, is the synthetic integer dataset with a Zipf distribution with $\eta = 1$, and $|N_{samp}| = 5 \cdot 10^7$.

For both this algorithm as well as the subsequent two, we note the runtime has two regions. When ϵ is small, the runtime is approximately linear in $|N_{samp}|$. As ϵ grows, it becomes the primary cost of algorithmic runtime and the runtime is linear in ϵ^{-1}.

Varying FPRs Across Layers. When allowing the α_i values to change across layers, we are faced with a non-convex optimization objective and constraint, even when fixing T_L (this can be seen by taking second derivatives). To perform optimization, at each checked value of N_f we first run the optimization using equal FPR across layers. We then polish the resulting filter by using

	Bloom, fixed α		Bloom, varied α		Cuckoo	
ϵ	EFPR	Time	EFPR	Time	EFPR	Time
10^{-2}	0.00175	$775\mu s$	0.00175	47 ms	0.00203	12 ms
10^{-3}	0.00173	$781\mu s$	0.00173	328 ms	0.00190	13 ms
10^{-4}	0.00172	1.01 ms	0.00172	2.9 s	0.00184	44 ms
10^{-5}	0.00172	3.7 ms	0.00172	26.7 s	0.00184	367 ms

Table 4.2: Optimization is efficient and tunably optimal

the gradient-free algorithm COBYLA [Powell, 1994] to modify the FPRs of each layer. While this method very occasionally achieves improvements over the fixed FPR per layer method, it generally does not, as seen in Table 4.2. Additionally, an alternative strategy of discretizing the search space for the FPRs at each layer and using the optimization routines described in Section 4.5.2.3 also did not in general improve upon the equal FPR per layer solution. Thus we view this final polishing as optional in the optimization of continuous FPR filters.

4.5.2.3 Inner Optimization: Fingerprint Based Filters

For fingerprint based filters, the discrete number of fingerprint bits makes search easier. The main idea of our approach is to use breadth first search expanding the number of fingerprint bits used at each layer, working two layers at a time: one positive and one negative. At each pair of layers, derived bounds on which possible fingerprint lengths can lead to an optimal solution of each layer are used, constraining the number of options expanded. Eventually, each search path terminates, either because its choices already created too many false positives, it used all the available space budget, or the number of queries which would reach the current layer of the chosen stack is less than ϵ. The full algorithm and its explanation are given in the Appendix .

Theoretically, the algorithm is guaranteed to return a filter with $EFPR \leq EFPR^* + \epsilon$, where $EFPR^*$ is the EFPR of the best possible filter satisfying all constraints. We can bound the runtime of the algorithm theoretically by $O(|N_{samp}| + \epsilon^{-3})$. Additionally, the proof of the runtime bound does not rely on several key optimizations of the algorithm, and the experimental run time of the algorithm behaves more like $O(\epsilon^{-1})$. Thus, the algorithmic run time is both tunable and efficient, as can be seen in Table 4.2 for Cuckoo Filters.

4.6 Incremental Construction and Adaptivity

So far we assumed that workload knowledge can be collected and that workloads are static or drift slowly. While this holds for many filter use cases such as URL Blacklisting [Kraska et al., 2018] and Web Indexing [Goodwin et al., 2017], there are many other applications where workloads change quickly, in which case continuously gathering workload knowledge is expensive. To address these use cases, we introduce Adaptive Stacked Filters (ASFs) which require knowledge about workload shape (such as how skewed they are), but do not require the gathering of a set of negative queries. Crucial to the design of ASFs is the idea of incremental construction, which allows ASFs to process queries immediately, learn frequent negatives during query evaluation, and gain benefits from stacking before finishing construction. Mirroring how we described Stacked Filters, we explain first the structure of ASFs given their parameters, then how to collect workload knowledge, and finally how to optimize their parameters.

Incremental Construction. ASFs start by constructing L_1 and use this to answer incoming queries until more layers are constructed. They also allocate an empty L_2. During query processing, when a false positive occurs, it is added to L_2. Then, when L_2 is full (in that it either hits its load factor for Cuckoo and Quotient filters or has half its bits set for Bloom filters), the ASF brings in the positive set, queries it against L_2 and adds the false positives on L_2 to a new L_3. Processing then continues using the layers up until L_3, gaining the benefits in terms of EFPR that come with extra layers. Additionally, construction on layers L_4 and L_5 can begin (if they exist), and uses the same procedure. Since N_f is captured during query processing, it does not need to be gathered before the construction of the ASF. This is the primary benefit of ASFs over Stacked Filters: they require only the workload shape (to figure out how big each layer should be) but not which values are important.

Rank-Frequency Workload Knowledge. To create ASFs, we need a rank-frequency distribution of the negative query workload. This is akin to the workload knowledge of Section 4.5.1, but instead of describing the query frequencies of actual values, the distribution describes the query frequencies of the value at each rank, where rank is itself defined by ordering elements' query frequencies. A classic example of this type of distribution is the Zipf distribution, which models how often the first and second most popular values occur without reference to the actual values.

The rank-frequency distribution can be calculated in many different ways. We do so using a set of past queries to make a smoothed histogram as described in Section 4.5.1. ASFs assume that the workload shape is relatively static even if the values are not, and under this case, ASFs rebuild themselves without performing a new analysis of the workload. For instance, YouTube video queries and other periodic workloads are a good example of a case where this holds: queries for popular videos on a given week follow consistent patterns even if which videos are popular changes each week [Cha et al., 2009]. In this case, an ASF being rebuilt and adapting to new frequent values knows what shape to take before it knows the new frequent values.

Optimizing Collection vs Exploitation. We optimize ASFs in two different ways, depending on the nature of the workload. The first uses the optimization procedures of base Stacked Filters assuming we pick the most frequently queried negatives and allocates the exact same filter. It then creates the filter incrementally during query processing instead of all at once.

The above process builds the best eventual ASF but can face many queries before achieving a fully built ASF. For this reason, we additionally create a second approach which assumes that ASFs are 3 layers and focuses on building a fully built ASF very quickly. This form of optimization takes as input an estimate of how long the filter will last and then chooses a number of queries to observe when building the second layer, denoted by N_o. The value of N_o determines the expected values of ψ and $|N_f|$:

$$E[\psi] = \sum_{x \in N_{samp}} f(x)(1-(1-f(x))^{N_o})$$
$$E[N_f] = (\ell \cdot N_o) + \sum_{x \in N_{samp}} (1-(1-f(x))^{N_o})$$

where here we recall that ℓ is the estimate of what portion of queries fall on values outside our sample. The optimization then weights the EFPR using just L_1 for N_o queries vs. the EFPR of the ASF using all 3 layers on the rest of the queries. To choose the best configuration, we perform grid search on N_o. At each value of N_o, we either calculate or estimate ψ and $|N_f|$, depending on the size of N_{samp}, and perform optimization of the three layers using discrete search for both continuous FPR filters and integer-length fingerprint filters (see Appendix Sec. B.5 for details).

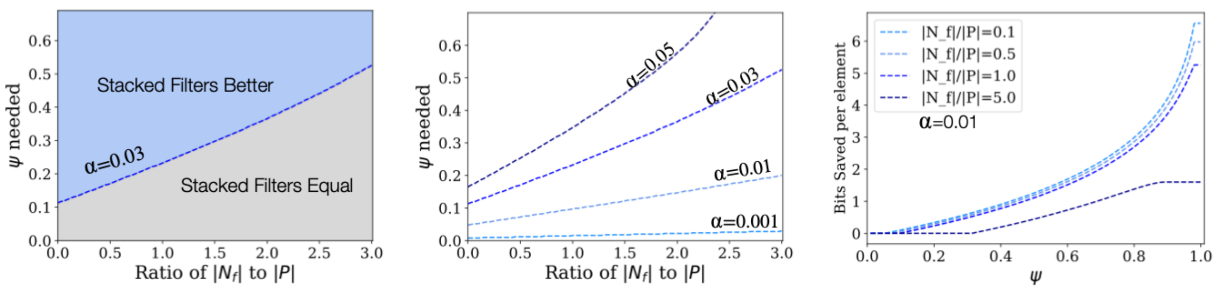


Figure 4.2: Analytical equations can predict both when Stacked Filters are better, and by how much.

Monitoring and Adapting. To maintain robust performance, if the elements in N_f become less frequently queried over time, this needs to be rectified. To address this, the ASF monitors its performance and initiates a rebuild whenever the FPR differs by more than 50% from its expected FPR. The ASF initially tries a rebuild assuming that the popularity of particular values has changed but not the rank-frequency distribution; the layers after the first of the ASF are dropped and the procedure for construction starts from L_1. If this does not fix performance, a remodeling of the workload happens, and the filter is re-optimized and rebuilt from scratch.

4.7 Better Size-FPR Tradeoffs

With Stacked Filters and their adaptive counterparts described fully, we ask the following critical question: when are Stacked Filters better than query-agnostic filters and by how much? In terms of their trade-off between EFPR and size, the following theorems answer this question using only the parameters of the workload: namely a choice of $|N_f|$ and $\psi(N_f)$. Along with each theorem, we present a visualization of its results, which can be used by systems designers to estimate the benefits of Stacked Filters on their workload a priori to spending the time to gather workload knowledge.

Each theorem holds exactly in the case that α is a continuous parameter for the base query-agnostic filter, and we discuss how the theorems apply to integer length fingerprint filters at the end of this section. The first theorem shows when a Stacked Filter is strictly better than a query-agnostic filter, as opposed to when a 1-layer filter is best.

Theorem 4.7.1. *Let the positive set have size $|P|$, let the distribution of our negative queries be D,* *and let α be a desired expected false positive rate. If there exists any set N_f, $\psi = \mathbb{P}_D(x \in N_f | x \in N)$, and $0 \leq k \leq \psi$ such that*

$$\frac{|N_f|}{|P|} \leq \frac{\ln \frac{1}{1-k}}{\ln \frac{1-k}{\alpha} + c} \cdot \frac{1-k-\alpha}{\alpha} - 1$$

then a Stacked Filter (optimized using Section 4.5 and given access to any N_f satisfying the constraint) achieves the EFPR α using fewer bits than a query-agnostic filter.

Figures 4.2a and 4.2b use this theorem to create visualizations of which workloads Stacked Filters are certainly better than query-agnostic filters. Figure 4.2a shows this for a desired EFPR of $\alpha = 0.03$; any workload with a set N_f such that $|N_f|, \psi(N_f)$ is above the line has a Stacked Filter which is strictly better than a Bloom filter. Figure 4.2b shows this trend for more alpha values. Even at a high desired α of 0.05, Stacked Filters cover a sizeable number of workloads; many workloads contain a negative set half the size of their positive set, and which contain 25% of all negative queries. As the desired EFPR decreases, Stacked Filters cover almost all real workloads; for instance at a desired EFPR of $\alpha = 0.01$, if $|N_f| = |P|$, then only 7% of negative queries need to be at values in N_f for the Stacked Filter to be more space efficient. At even lower values, the amount needed becomes negligible and almost any workload sees improvements.

Estimating Space Savings from Stacked Filters. Using the optimization routines of the prior sections and given values for $|N_f|$ and $\psi(N_f)$, it becomes possible to estimate the space savings of using a Stacked Filter as compared to a query-agnostic filter for any desired EFPR. Namely, we can optimize the size of a Stacked Filter with equal FPRs across layers while requiring that its EFPR is less than a query-agnostic filter:

$$s(\alpha) - \min_{\alpha_L} s(\alpha_L)(\frac{1}{1-\alpha_L} + \frac{|N_f|}{|P|}\frac{\alpha_L}{1-\alpha_L})$$
$$s.t. \qquad \alpha_L \in (0, \frac{\alpha}{1-\psi}]$$

Figure 4.2c uses this to graph how a combination of $\frac{|N_f|}{|P|}$ and ψ produces a reduction in filter size using Stacked Bloom Filters at a desired EFPR of $\alpha = 0.01$. For each fixed value of $\frac{|N_f|}{P}$, the graph contains three parts: in the first part, it is not advantageous to build Stacked Filters and a query-agnostic filter is built. For all values of $\frac{|N_f|}{|P|}$, this is a small area and covers workloads with no frequently queried negative elements. In the second part, Stacked Filters have superlinear improvement in ψ, with improvement starting at 0 bits per element saved and going up to 7 bits per element saved. At the tail end of the graphs, the improvement stops even as ψ increases. This is the point where $\frac{\alpha}{1-\psi}$ crosses the minimal α_L value for size. After, the Stacked Filter can choose larger false positive rates at each layer while having the same EFPR as the query-agnostic filter, but this larger α_L value increases size. Instead, the Stacked Filter keeps the minimal α_L value for size and the Stacked Filter produces both a space benefit and has lower EFPR than the input α.

Integer Length Fingerprint Filters. The above equations and theory assumed that all FPRs were possible at each layer. For integer length fingerprint filters, this is not the case and the theory does not hold exactly; however, the general trajectory remains the same. Additionally, experimentally the results for integer length fingerprint filters are often better than the continuous FPR approximations suggest. This is because query-agnostic filters also suffer from limitations on the FPRs they can choose; often given more size as a budget there isn't enough space to add a full bit for every positive element. In these cases, Stacked Filters can often make use of this space to build layers deeper in the stack, and the added flexiblity of being able to use space on any layer in the stack provides additional improvements over the theory above.

4.8 Experimental Analysis

We now experimentally demonstrate that Stacked Filters offer better false positive rates compared to query-agnostic Filters for the same size, or they offer the same false positive rate at a smaller size. We also show that Stacked Filters are more computationally efficient, robust, and are more generally applicable than classifier-based filters while offering similar false positive rates and sizes.

Filter Implementations. All filters use CityHash as the hash function [Pike and Alakuijala, 2011]. The Counting Quotient Filter (CQF) and Cuckoo Filter (CF) implementations are taken from the original papers [Pandey et al., 2017, Fan et al., 2014]. In the original implementation, CQF is constrained to have the filter size be a power of two to allow for operations such as resizing and merging. This is not relevant to our testing, so we removed this restriction. For CF, the implementation provided only supports certain signature lengths, so we implemented a fix to allow

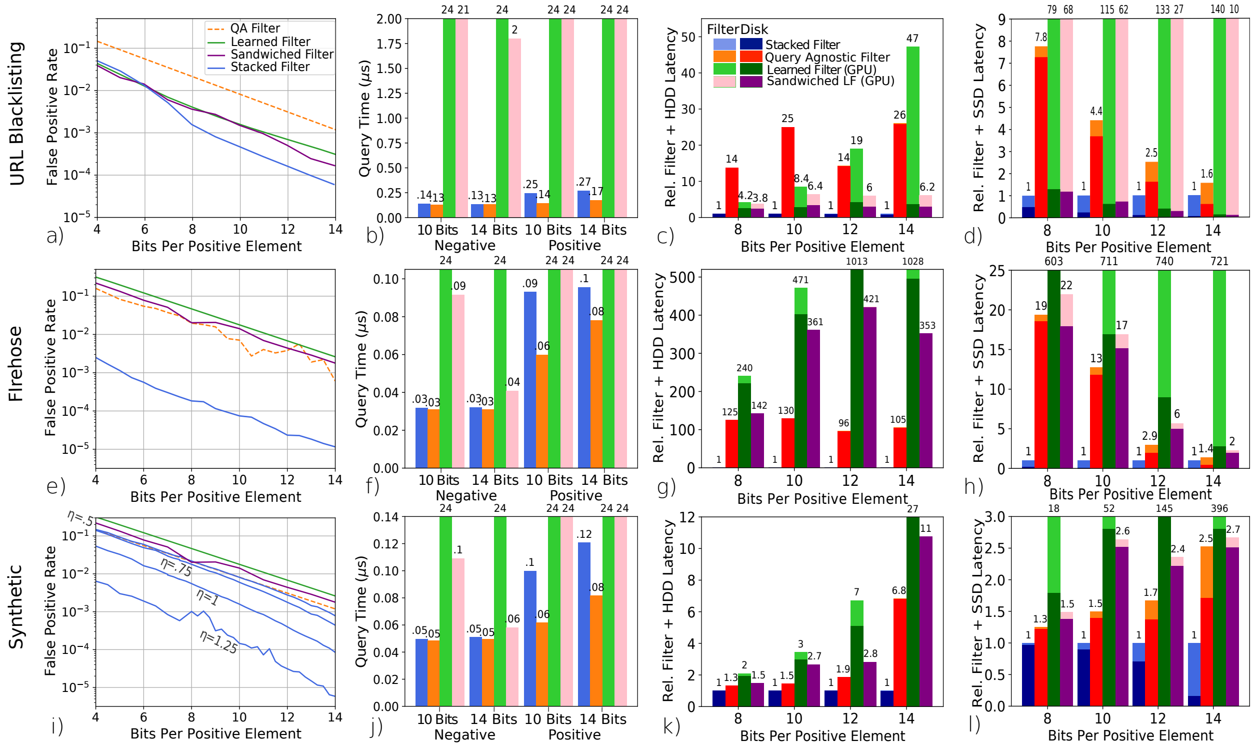


Figure 4.3: Stacked Bloom Filters achieve a similar EFPR to Learned Bloom Filters while maintaining a 170x throughput advantage and beat both query-agnostic and learned filters in overall performance on typical workloads.

all integer signature lengths. For classifier-based filters, we use Learned Bloom Filters [Kraska et al., 2018] with text data using a 16 dimensional character-level GRU as in the original paper, and integer data using a shallow feed-forward neural network. Additionally, we compare with Sandwiched Learned Filters (SLF) [Mitzenmacher, 2018], which uses the same model as the Learned Filter but has a query-agnostic pre-filter as well as a backup filter (Bloom filters). For Stacked Filter layers we use the same implementations as in the query agnostic filters. For most experiments we use Bloom Filters and we refer to the filter as Stacked Bloom Filter but we also show results with other filters.

Experimental Infrastructure. All experiments are run on a machine with an Intel Core-i7 i7-9750H (2.60GHz with 6 cores), 32 GB of RAM, and a Nvidia GeForce GTX 1660 Ti graphics card. Each experimental number reported is the average of 25 runs.

Datasets. We use three diverse datasets:

1. *URL Blacklisting*: The URL Blacklisting application was used to introduce Learned Filters

[Kraska et al., 2018]. As the dataset in [Kraska et al., 2018] is not publicly available, we instead use two open-source databases, Shalla's Blacklists [sha] as a positive set of dangerous URLs, and the top 10 million websites from the Open Page Rank Initiative[dom] as a negative set of safe URLs, with the probability of querying a safe URL proportional to its PageRank.

2. *Packet Filtering*: Packet filtering is a common application for filters and was used to evaluate Counting Quotient Filters [Pandey et al., 2017]. Following their lead, we use the benchmark Firehose [Anderson and Plimpton, 2015] which simulates an environment where some subset of packets are labeled suspicious and need to be filtered. The benchmark is run under its default settings.
3. *Synethetic Integers*: To more finely control experimental settings, we also use synthetic data. The data set consists of 1 million positive elements using randomly generated integer keys. Negative queries on the dataset come from a set of 100 million negative elements, also with randomly generated keys, and follow a Zipfian distribution. The skew of the negative query distribution is a controlled parameter η taking values between 0.5 and 1.25. For all experiments and graphs where η is not listed, $\eta = 0.75$.

For each dataset, we simulate having incomplete information about the query distribution by giving Learned, Sandwiched Learned, and Stacked Filters only the higher frequency half of the negative set for training. We also varied this amount from between 10% to 80% of the training data and saw the same relative results.

4.8.1 Evaluating Total Filter Performance

The overall performance of a filter can be broken down into two pieces, 1) the overhead incurred by the filter checks and 2) the cost of unnecessary operations incurred by false positives. While 1) is a characteristic of just the filter, 2) depends both on the FPR of the filter and the cost of the operations the filter protects against. Thus, we fix memory and measure FPR and probe (computation) time. We then translate these two metrics into total filter performance for the common case of protecting against base data accesses on persistent hardware using a slower HDD (favoring lower FPRs) and a faster NVMe SSD (favoring lower computational rates).

Workload Knowledge Improves FPRs. Figures 4.3a, e, and i show the false positive rates for all filters across all three datasets. Here we use Bloom filters for both the query-agnostic filter and the Stack Filter. We see that Stacked Bloom Filters is the clear winner across all workloads. Learned Filters are closer for the URL Blacklisting scenario which is a favorable scenario for learning but still Stacked Filters provide a better FPR for most memory budgets. For other workloads, Learned Filters give similar or slightly worse FPR than traditional query-agnostic filters while Stacked Filters provide a drastic benefit.

Across all three workloads, the negative query distribution has frequently queried elements and so Stacked Filters can find a “small" set of negative elements that contain a large portion of the query workload (we need only N_f to not be dramatically larger than P). Under these conditions, deeper stacks have only marginal overhead compared to a single layer and so Stacked Filters allocate almost all their space budget to the first layer. This way, they achieve essentially the same FPR as query-agnostic filters on infrequent negatives and at the same time, eliminate all false positives from frequently queried negatives. Stacked Filters improve upon the FPR of query-agnostic Bloom Filters by 5-100×, as seen in Figures 4.3a, e, and i. Alternatively, Stacked Filters can reduce their size significantly while achieving the same FPRs as query-agnostic filters.

Learned and Sandwiched Learned Filters’ performance depends heavily on the dataset. With URL Blacklisting (Fig. 4.3a) where keys have semantic meaning and are correlated with their appearance in the positive or negative set, Learned and Stacked filters achieve similar results. When the keys have less semantic meaning such as in Firehose (Fig. 4.3e) or the synthetic integer data (Fig. 4.3i), Learned Filters’ performance degrades and becomes worse than a standard Bloom Filter. Overall, the evaluation of Learned Filters depends on 1) how correlated keys are with their positive/negative set membership, and 2) how complicated their decision boundaries are for the dataset (this affects computational performance as well). Figure 4.3a shows that even on tasks that are considered good fit for Learned Filters, their performance can be matched by Stacked Filters.

Hash-Based Filters Dominate Classifier-Based Filters Computationally. Figures 4.3b, f, and j depict the computational performance of each filter. Noticeably, Learned Filters have computational performance that is orders of magnitude slower than hash-based filters, with the

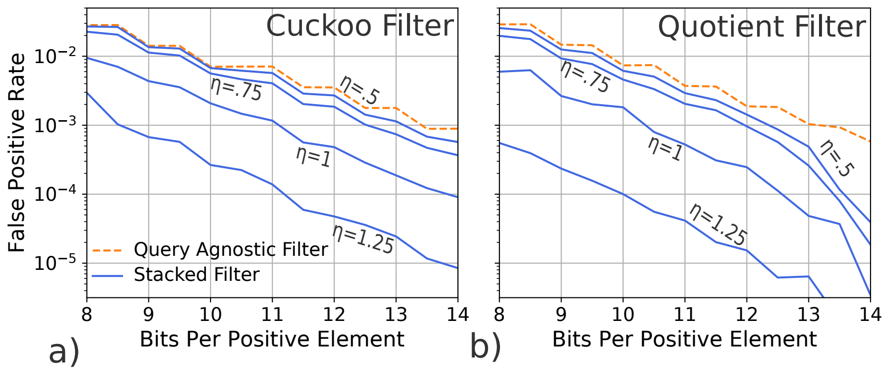


Figure 4.4: Stacking provides robust performance benefits across a variety of underlying filter types.

difference between 90-190×. In comparison, Stacked Filters have computational performance more in line with query-agnostic filters, with the difference between 1-2×. For queries on negative elements, the Bloom Filter and Stacked Bloom Filter have essentially identical computational cost, and for queries on positive elements, the Stacked Filter probe is about 1.5× the cost of the Bloom Filter probe. For Sandwiched Learned Filters, their performance is a weighted combination of hash-based filters and the classifier; overall, they are at least 3× more expensive and can be as much as 90× more computationally expensive.

Stacked Filters Maximize Overall Performance. As Figures 4.3 c-g-k and d-h-l show, Stacked Filters strike the best balance between decreased false positive rates and affordable computational speeds, resulting in the best overall performance across both hard disk and SSD. Compared to query-agnostic filters, both have fast hash-based computational performance but Stacked Filters have significantly fewer false positives; as a result, they provide total workload costs which are 1.4-130× lower. In comparison to the classifier-based Filters, Stacked Filters are better or equal in terms of false positive rates and better by orders of magnitude in computational performance, resulting in 1.5-1028× lower total workload costs.

4.8.2 Stacking Improves Diverse Filter Types

Figures 4.4a, and b show that Stacked Filters benefits generalize across diverse filter types used for the stacked layers. More specifically, Figure 4 shows the performance of Stacked Cuckoo

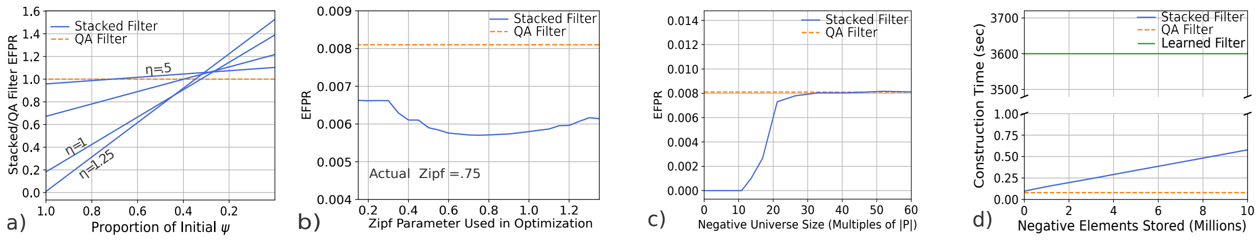


Figure 4.5: Stacked Filters are robust to workload shifts, skew, noisy data, and can be rebuilt quickly.

and Stacked Quotient Filters on the synthetic integer dataset. This is the same experiment as for Stacked Bloom Filters in Figure 4.3i. Collectively these three graphs all show the same benefits, which is that Stacked Filters achieve lower false positive rates compared to their query-agnostic counter-parts as workloads are more skewed. This is because more skewed workloads query their frequent negatives more often and because at lower FPRs the first layer in a Stacked Filter culls almost all elements, making subsequent layers in the Stack cheaper to build (as can be seen in Equation (2)). Further, while not shown here, the computational costs of Stacked Cuckoo and Stacked Quotient Filters follow the same patterns as seen in Figures 3b, f, and j, leading to similar benefits in terms of total throughput.

4.8.3 Stacked Filters are Workload-Robust

We now demonstrate that Stacked Filters retain the robustness of query-agnostic filters, bringing an additional benefit over classifier-based filters. We focus on two facets of robustness: maintaining performance under shifting workloads and providing utility for a variety of workloads. We use the synthetic integer dataset with size $s = 10$, and Bloom filters for both the query-agnostic and the Stacked filter.

Robust to Workload Shifts. Figure 4.5a shows how Stacked Filters' performance adapts to workload changes with diverse values for η. We vary the workload by reducing the value of ψ from its initial value to a value of 0. This captures scenarios where the frequently queried negative values are changing over time, and so Stacked Filters are no longer optimized for the most frequently queried negatives. For Stacked Filters, regardless of the skew of their initial distribution, drastic workload shifts are needed to become worse than query-agnostic filters, with query-agnostic filters being better only after the frequently queried negative set loses more than 60% of its initial set. Even in the extreme case that every frequently queried element from when the Stacked Filter was built is no longer queried ($\psi = 0$), the Stacked Filter is never more than 50% worse than a query-agnostic filter.

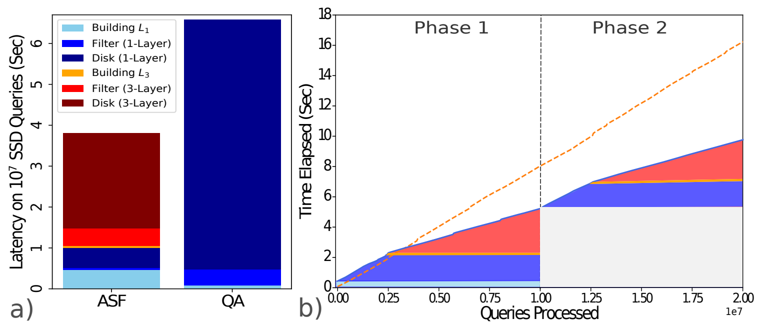
Robust to Workload Misspecification. Figure 4.5b shows the behavior when the sample queries used to build the filter come from a different distribution than the future workload. Here the queries come from the integer dataset with $\eta = 0.75$, however, during workload modeling, we used η values from 0.15 to 1.35. Figure 4.5b shows that even when the modeled workload is significantly different from the true workload, the Stacked Filter retains most of its performance and outperforms query-agnostic filters.

Robust to Workload Type. Because Stacked Filters rely on taking advantage of frequently queried negative values, uniform workloads are the most difficult ones. However, if $|N|$ is relatively small, every negative element is still frequently queried. Figure 4.5c shows that Stacked Bloom Filters outperform query agnostic Bloom filters when $|N|$ is a reasonable multiple of $|P|$, anywhere up to $25\times$. Since the uniform distribution is the worst distribution possible for Stacked Filters, this shows that Stacked Filters are better than query-agnostic filters for all small universe sizes.

Easy to Reconstruct. While Stacked Filters are effective under shifting workloads, they need to be rebuilt to regain their best performance. Figure 4.5d shows that the construction cost grows slowly with the number of the frequently queried negatives, is significantly faster than classifier-based filters, and is comparable to query-agnostic filters. Because Stacked Filters can be reconstructed quickly, they can handle periodic bulk updates. Such a strategy is key for handling workloads with dynamic positive data.

4.8.4 Incrementally Adapting to Workload Shifts

We now show that Adaptive Stacked Filters (ASFs) are capable of adapting quickly to changing workload patterns improving on the (good) worst case guarantees of base Stacked Filters. We use the synthetic integer dataset with Zipf parameter 1, let $s = 10$ bits per element, and use Bloom



c)

	ASF 1-Layer	QA Filter	ASF 3-Layer	Static SF
FPR	.0101	.0087	.0028	.0017

Figure 4.6: High performance without workload knowledge.

Filters for both the query-agnostic and Stacked Filter. The ASF is optimized using the 3-layer approach.

Fast Incremental Construction. Figure 4.6a shows the performance of an ASF and a query-agnostic filter on a static workload of 10 million queries, with the filters protecting against data accesses to an SSD. The height of each bar shows the total time to process the queries, and the colors indicate the breakdown of how that time is spent. Overall, the ASF results in better performance over the workload compared with the query-agnostic filter, as its gains in FPR after the 3rd filter is built more than compensate for the cost to build L_3 and its slightly worse performance when using only L_1.

Adapting to Shifting Workloads. Figure 4.6b shows that ASFs work well for shifting workloads. Phase 1 of the experiment replicates the previous experiment; then, in phase 2, the query frequency distribution remains a Zipf with parameter $\eta = 1$ but the popularity of the elements is flipped so that the previously least popular element is now the most popular and vice versa. The figure details how long the ASF and query-agnostic filter take to process each workload. Additionally, the colored bars show what phase the ASF is in during query processing at the time of each query.

While phase 1 shows as before that the ASF performs well on static workloads, phase 2 shows that the ASF can adapt to shifts in workload. The ASF does so by recognizing at the start of phase 2 that a shift has occurred and signaling a rebuild. Then, performance continues as in a static phase: the ASF learns the new workload pattern quickly by incrementally building layer 2, then builds layer 3 and capitalizes on the reduced false positive rate once that occurs. Ultimately, across both phases, ASFs process the workload 1.6× faster than query-agnostic filters.

Breaking Down ASF Performance. Figure 4.6c breaks down the benefits of ASFs and compares them with static Stacked Filters. The table repeats a trend seen throughout the paper: the first layer of a Stacked Filter is nearly as performant as a query-agnostic filter. Then, as the ASF builds its 3rd layer its FPR drops to nearly 1/4 of its previous value and is 3× more performant than the query-agnostic filter. Finally, we see that compared to a static Stacked Filter on phase 1, ASF is about 1.6× worse, so if workloads are sufficiently static, then a static Stacked Filter is best.

4.9 Conclusion

With the experimental evaluation complete, we recap Stacked Filters. We started with the realization that no parameterization led query-agnostic filters to exhibit sub-optimal FPR, but that the rich parameterization of the workload used by learned filters required a classifier which lead to their designs being computationally slow and not robust. We then used the framework of Cerebral Data Structures to build out an alternative parameterization that allowed for capturing workload knowledge without the use of a classifier, and used this to create a filter with the FPR vs. size tradeoffs of learned approaches while maintaining the computational benefits and robustness of traditional filters. The end result is a filter that has excellent balance between filters four main metrics of FPR, size, robustness, and computational speed.

Chapter 5

Cerebral Scans: Column Sketches

This chapter introduces our third example of designing cerebral data structures, Column Sketches, which looks at how estimating the distribution of values in a workload allows for the creation of (nearly) uniform representations so that every bit is equally informative, and how this then interacts with current approaches to hardware-conscious scans over unsorted arrays. The end result is a data structure which, unlike prior methods, provides robust and efficient scan performance regardless of data distribution, data ordering, or the selectivity of the predicate applied. We start by motivating work on scans over unordered array and reviewing prior approaches to optimizing scans in database systems before diving into Column Sketches.

5.1 Robust Scans through Robust Data Representations

5.1.1 Analytical Database Scans and their Optimizations

Base data access and methods for predicate evaluation are of central importance to analytical database performance. Indeed, as every query requires either an index or full table scan, the performance of the select operator acts as a baseline for system performance. Because of this, there exists a myriad of methods and optimizations for enhancing predicate evaluation [Feng et al., 2015, Johnson et al., 2008, Li and Patel, 2013, Metzger et al., 2005, Pirk et al., 2014, Polychroniou et al., 2015, Sidirourgos and Kersten, 2013, Willhalm et al., 2013]. Despite the large volume of work done,

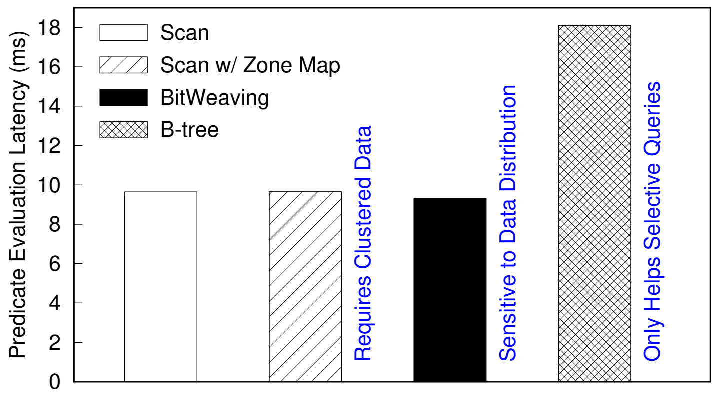


Figure 5.1: For certain workloads, no state of the art methods provide any significant performance benefit over a scan.

existing access methods each have situations in which they perform suboptimally. Figure 5.1 shows an example where different classes of access methods from traditional indexing such as B-trees to scan accelerators such as Zone Maps [Metzger et al., 2005] and BitWeaving [Li and Patel, 2013] do not bring any improvement over a plain scan. We use Figure 5.1 throughout this section as we discuss the performance characteristics of state-of-the-art indexing and scan accelerator methods, and use their performance to motivate Column Sketches.

A long term staple of database systems, traditional secondary indices such as B-trees localize data access to tuples of interest, and thus provide excellent performance for queries that contain a low selectivity predicate [Comer, 1979]. However, once selectivity reaches even moderate levels, the performance of B-trees is notably worse than other methods. Figure 5.1 shows such an example with low selectivity at 3%. However, as mentioned in Section 2.4, traditional indexes look at data in the order of the domain, not in the order of the table. As a result, their output leaves a choice: either sort the output of the index by the order of the table or continue through the rest of query execution looking at values out of order. This mismatch between the natural data ordering for the rest of the table and that of the index limits the effectiveness of traditional indexes as more data is selected, with scans outperform B-trees for query selectivities as low as 1% [Kester et al., 2017].

Lightweight indexing techniques make an impact on scan performance by skipping data while doing an in-order scan. Across techniques such as Zone Maps, Column Imprints, and Feature-Based Data Skipping, the idea is to keep small mounts of metadata which exploit natural clustering properties and use this to skip over groups of data. For instance, Zone Maps store min and max values per block of data and can use this to decide if an entire block of data qualifies if the query value is outside of the (min,max) range. This approach works when data is clustered, and Column Imprints actually provide a parameter of the workload which measures how clustered data is (they call this metric "column entropy"). This approach works very well in the right circumstances, when data to be clustered but provides no benefit when it is not, as in the case of Figure 5.1. Thus, while useful, more parameters of the workload beyond column entropy are needed to provide good performance across workloads.

Early pruning methods such as Byte-Slicing [Feng et al., 2015], Bit-Slicing [Li and Patel, 2013, O'Neil and Quass, 1997], and Approximate and Refine [Pirk et al., 2014] techniques are ways to speed up in-order scans over data by physically decomposing individual values. After physically partitioning the data, each technique takes a predicate over the value and decomposes the predicate into conjunctions of disjoint sub-predicates. As an example, checking whether a two byte numeric value equals 100 is equivalent to checking if the high order byte is equal to 0 and the lower order byte is equal to 100. These techniques then evaluate the predicates in order of highest order bit(s) to lowest order bit(s), skipping predicate evaluation for predicates later in the evaluation order if groups of tuples in some block are all certain to have qualified or not qualified. The key to these techniques is that data is skipped if higher order bits are informative, whereas if data is skewed such that higher order bits are uninformative, then all bits of data may need to be read anyway. This is what happens in the case of Figure 5.1; the high order bits are biased towards zero, enabling very little pruning. As a result early pruning brings no significant advantage over a traditional scan.

5.1.2 Cerebral Database Scans

We now use our framework of Cerebral Data Structures to improve the scan operator in a context-specific manner. We start by noting that the main metrics for scans are the speed of its operations, the robustness of scan behavior, the ability to read in new data, and the memory taken up by whatever auxiliary structures are created (step 1). For scans, depending on use case, any metric can be prioritized and so what is important is to push forward the trade-off curve between memory overhead, ingestion speed, scan performance, and robustness. When viewing past approaches to scans, we focus on early-pruning methods and look at two aspects for improvement: the physical organization the data (how data is decomposed), and the representation of each data item (step 2a).

We focus in our approach to scans on how to make each bit informative in early pruning methods with a specific focus on the evaluation of range and equality predicates as these are the most common predicates in relational databases. Our key intuition comes from the fact that for each bit to equally informative, we want a uniform distribution over values (so that each bit is equally likely to be a 1 or 0, given the values of all higher order bits). The question is how to make this occur for a column of data values while preserving the order of a given domain, the ability to reconstruct data values, and while making the representation work well with modern hardware. Our solution here is to turn to lossy compression, and to apply a non-injective map on a value-by-value basis to create an auxiliary column of codes which is a "sketch" of the base data. Then, predicate evaluation is broken up into (1) a predicate evaluation over the sketched data, and, if necessary, (2) a predicate evaluation over a small amount of data from the base data.

The key to making this approach work is to parameterize the workload in a way that allows for creating a map which creates an efficient auxiliary column for scan performance. We do this by estimating the CDF of the data values for a column (step 2b). Using the probability integral transform, it can be shown that applying F to a continuous distribution of data values produces a uniform random variable. This can be roughly modified for discrete distributions in a way that every bit of the resulting variable is useful for predicate evaluation. The estimation of the CDF along with our approach to decomposing data value then allows us to develop concrete metrics which bound the amount of data that needs to be read during scans for any query and allows efficient and robust query performance (step 2b). Additionally, we can show via our approach to lossy compression that the overhead in memory footprint is small, and that the lossy compression scheme is amenable to fast ingestion of data values (step 2b). Finally, to make this data structure happen in practice, we provide estimators for the CDF (step 3) and then simply apply this approximate CDF to the column of base data (or incoming new data values) to produce the required auxiliary column (step

4). We now build out all these ideas in more detail over the coming sections.

Contributions. The major contributions of the Cerebral Data Structures approach to scans over unordered arrays are:

1. We introduce the Column Sketch, a data structure for accelerating scans which uses lossy compression to improve performance regardless of selectivity, data-value distribution, and data clustering (Sec. 5.2).
2. We show how lossy compression can be used to create informative bit representations while keeping memory overhead low (Sec. 5.3).
3. We provide algorithms for efficient scans over a Column Sketch (Sec. 5.4), give models for Column Sketch performance (Sec. 5.5), and show how Column Sketches can be easily integrated into modern system architectures (Sec. 5.6).
4. We demonstrate both analytically and experimentally that Column Sketches improve scan performance by 3×-6× for numerical data, 2.7× for categorical data, and improve upon current state-of-the-art techniques for scan accelerators (Sec. 5.7).

5.2 Column Sketches Overview

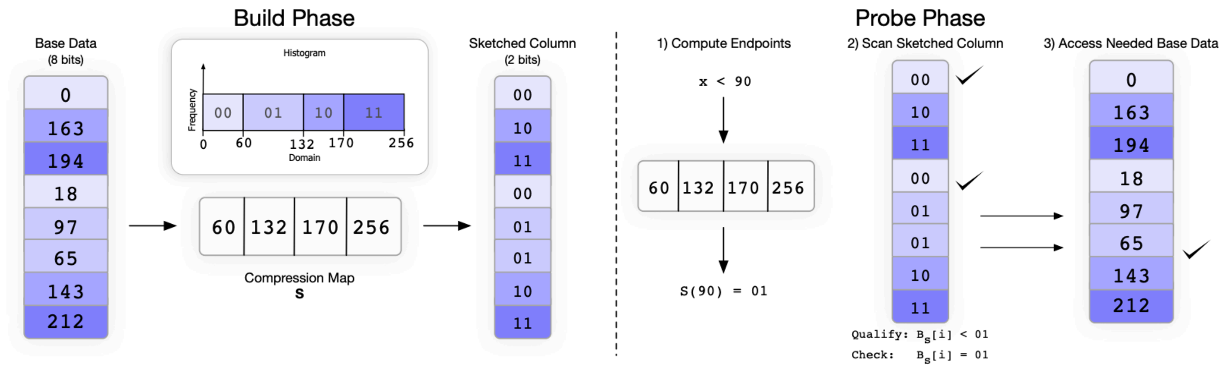


Figure 5.2: Column Sketches use their compression map to transform (possibly compressed) values in the base data to smaller code values in the sketched column. These codes are then used to filter most values in the base data during predicate evaluation.

We begin with an illustrative example to describe the main idea and the storage scheme. For ease of presentation we use a simple lossy compression function in this example. Then, Section

5.3 discusses in detail how to design lossy compression functions for robust and efficient predicate evaluation.

Supported Base Data Format. The only requirement for the base data is that given a position i and base attribute B, that it be able to produce a value $B[i]$ for that position. Column Sketches work over row, column-group, or columnar data layouts, with the main body of this paper focusing on Column Sketches over columnar data layouts. Appendix C.1 discusses alternative layouts. As is common in state-of-the-art analytic systems, all base columns of a table are assumed to be positionally aligned and thus positions are used to identify values of the same tuple across columns [Abadi et al., 2013]. For numerical data types and dictionary encoded string data, the base data is an array of fixed-width values, with the value of position i at index i in the array. For unencoded variable length data such as strings there is one level of indirection, with an array of offsets pointing into a blob data structure containing the values.

Column Sketch Format. A Column Sketch consists of two structures. The first structure in a Column Sketch is the compression map, a function denoted by $S(x)$. An example compression map is shown in the middle of Figure 5.2a. The second structure is the sketched column, shown on the right side of Figure 5.2a. The term Column Sketch refers to the joint pairing of both the compression map and the sketched column.

(1) Compression Map. The compression map S is stored in one of two formats. If S is order-preserving, then we call the resulting Column Sketch order-preserving and the compression map is stored as an array of sorted values. The value in the array at position i gives the last element included in code i. For example, if position $i-1$ holds the value 1000 and position i holds the value 2400, then code i represents values between 1001 and 2400. In addition to the value at index i, there is a single bit used to denote whether the code is "unique". Unique codes are discussed in Section 5.3.

For non-order preserving Column Sketches, the function S is composed of a hash table containing unique codes and a hash function. In this format, frequent values are given unique codes and stored in the hash table. Infrequent values do not have their codes stored and are instead computed as the output of a (separate) hash function.

(2) Sketched Column. The sketched column B_s is a fixed width and dense array, with position i storing the output of the function S applied to the value at position i of the base data. To differentiate between values in the base data and the sketched column, we will refer to the values in the base data as simply values and the values in the sketched column as codes or code values.

Example: Building & Querying a Column Sketch. Consider the example shown in Figure 5.2, where we use the function S, defined by the array in the middle, to map from the 8 bit unsigned integers I_8 to the 2 bit unsigned integers I_2. S is order preserving and so it has the following properties:

1. for $x, y \in I_8$, $S(x) \neq S(y) \Rightarrow x \neq y$
2. for $x, y \in I_8$, $S(x) < S(y) \Rightarrow x < y$

Furthermore, S produces an output that is fixed-width (two bits) and assigns an equal number of values in the base data to each code.

We use S to build a smaller sketched column from the base data. For each position i in the base attribute B, we set position i in the sketched column to $S(B[i])$. The sketched column is $\frac{1}{4}$ the size of the original column, and thus scanning it takes less data movement. As an example, consider the evaluation of a query with the predicate WHERE $B < x$. Because S is order preserving, a Column Sketch can translate this predicate into $(B_s < S(x))$ OR $(B_s = S(x)$ AND $B < x)$.

To evaluate a predicate, the Column Sketch first computes $S(x)$. Then, it scans the sketched column S(B) and checks both $S(B) < S(x)$ and $S(B) = S(x)$. For values less than $S(x)$, their base value qualifies. For values greater than $S(x)$, their values in the base data do not qualify. For values equal to $S(x)$, their base value may or may not qualify and so we evaluate $B < x$ using the base data. Algorithm 1 depicts this process.

Figure 2 shows an example. Positions 1 and 4 in the sketched column qualify without seeing the base data. Positions 5 and 6 need to be checked in the base data, and of these two, only position 6 qualifies. In the example, the Column Sketch needs to go to the base data twice while checking 8 values. This is explained by the small number of bits in the compressed codes. In general, each code in a Column Sketch has a relatively equal number of values, and a Column Sketch needs to check the base data whenever it sees the mapped predicate value $S(x)$. As a result, we expect to

Algorithm 5 Select where $B < x$

```
Input: $S$ is a function which is order preserving, B is the base attribute, $B_s$ is the sketched column
1: for $i = 0$ to B.size do
2:   if $B_s[i] < S(x)$ then
3:     write position i to result output
4:   else if $B_s[i] == S(x)$ then
5:     if $B[i] < x$ then
6:       write position i to result output
```

need to go to the base data once for every $2^{\#\text{bits}}$ values. Thus, for larger code sizes of 8 or 16 bits, the Column Sketch scan will only need to check the base data for one out of every 2^8 or 2^{16} values.

The previous example gives the high level idea behind Column Sketches. For ease of presentation we did the example with a simple compression map and scan algorithm. In the rest of the paper, we build on the logical concepts covered here and show how Column Sketches use these concepts to deliver robust, efficient performance.

5.3 Constructing Compression Maps

We now show how to construct compression maps for a Column Sketch. By definition, this map is a function from the domain of the base data to the domain of the sketched column. To go over compression maps, we discuss their objectives in Section 5.3.1, give guarantees of their utility in Section 5.3.2, and discuss how to build them for numerical and categorical attributes in Sections 5.3.3 & 5.3.4.

5.3.1 Compression Map Objectives

The goal of the compression map is to limit the number of times we need to access the base data, and to efficiently support data modifications. To achieve this, compression maps:

(1) Assign frequently seen values their own unique code. When checking the endpoint of a query such as $B < x$, a Column Sketch scan needs to check the base data for code $S(x)$. If x is a value that has its own code (i.e. $S^{-1}(S(x)) = \{x\}$), then we do not need to check the base data and can directly answer the query through only the Column Sketch. This property holds for both range predicates and equality predicates.

To achieve robust scan performance, we identify frequent values and give them their own unique code. As a simple example to see why this is critical for robust performance, if we have a value that accounts for 10% of tuples and it has a non-unique code, then predicates on this value's assigned code need to access the base data a significant number of times. Because accessing any item of data in a cache line brings the entire cache line to the processor, accessing 10% of tuples is likely to make performance similar to a traditional scan. Thus, we identify the frequent values so that we can limit the amount of base data we touch for any predicate.

(2) Assign non-unique codes similar numbers of values. The reasoning for this is similar to the reasoning for why frequent values need unique codes. We assign each non-unique code a relatively even and small portion of the data set so that we need only a small number of base data accesses for any scan.

(3) Preserve order when necessary. Certain attributes see range predicates whereas others do not. For attributes which see range predicates, the compression map should be order-preserving so that range queries can be evaluated using the Column Sketch.

(4) Handle unseen values in the domain without re-encoding. Re-encoding should be a rare and on-demand operation. By nature of being lossy, lossy compression means new values are allowed to be indistinguishable from already occuring values. Thus, provided we define our encoding smartly, new values do not require a Column Sketch to be re-encoded. For ordered Column Sketches to not need re-encoding, there cannot be consecutive unique codes. For instance, if S assigns the unique codes i to "gale" and $i+1$ to "gate", then input strings such as "game" have no code value. Changing the code for "gate" to be non-unique solves this problem. For unordered Column Sketches, every unseen value has a possible value as long as there exists at least one non-unique code.

(5) Optional: Exploit Frequently Queried Values. Exploiting frequently queried values can give extra performance benefits; however, unlike frequent data values, identifying frequent query values makes query performance less robust. We focus on describing how to achieve efficient and robust performance for any query in the main part of the paper, and include details on exploiting frequently queried values in Appendix C.4.

5.3.2 Bounding Base Data Accesses

The following two theorems hold regarding how we can limit the number of values assigned to non-unique codes.

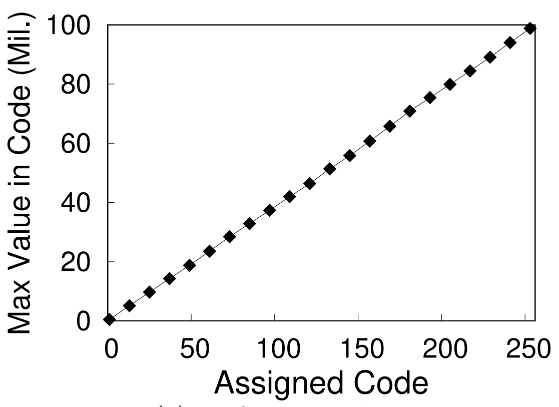
Theorem 5.3.1. *Let X be any finite domain with elements $x_1, x_2,$ $\ldots, x_n$ and order $x_1 < x_2 < \cdots < x_n$. Let each element x_i have associated frequency f_i with $\sum_{i=1}^{n} f_i = 1$. Let Y be a domain of size 256 and have elements $y_1, y_2, \ldots, y_{256}$. Then there exists an order-preserving function $S : X \to Y$ such that for each element y_i of Y, either $\sum_{x \in S^{-1}(y_i)} f_x \leq \frac{2}{256}$ or $S^{-1}(y_i)$ is a single element of X.*

Proof. We consider the greedy strategy for assigning codes to values of X. Assign $S(x_1), S(x_2), \ldots,$ $S(x_{c-1})$ to y_1, where c is the first number such that $\sum_{i=1}^{c} f_i > \frac{1}{128}$. Then let $S(x_c) = y_2$. Continue assigning y_3 and y_4, y_5 and $y_6, \ldots, y_{255}$ and y_{256} in a similar fashion, assigning codes to the odd y_i until the next code would make $\sum_{x \in S^{-1}(y_i)} f_x \geq \frac{2}{256}$ or until we have assigned x_n a code. It is clear that for all odd i, $\sum_{x \in S^{-1}(y_i)} f_x \leq \frac{2}{256}$ and for all even i, $S^{-1}(y_i)$ is a single element. Furthermore, by y_{256} we are guaranteed to have seen all x_n since for each pair (y_i, y_{i+1}) up until we reach x_n, we have $\sum_{x_j \in S^{-1}(y_i) \cup S^{-1}(y_{i+1})} f_j > \frac{2}{256}$. □

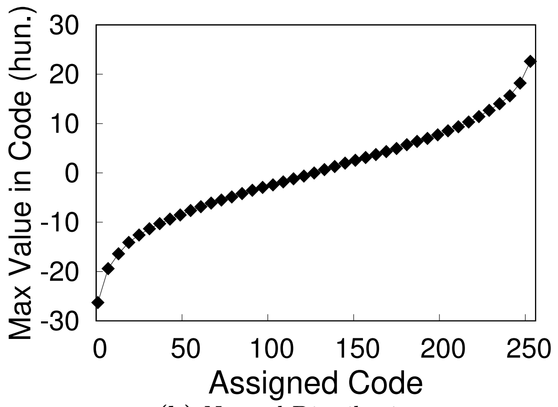
This theorem for an order preserving function then implies the result holds for a non-order preserving function as well.

Corollary 1. *Let X be any finite domain with elements $x_1, x_2, \ldots, x_n$ and let each element have associated frequency $f_1, f_2, \ldots, f_n$ such that $\sum_{i=1}^{n} f_i = 1$. Let Y be a domain of size 256 and have elements $y_1, y_2, \ldots, y_{256}$. Then there exists a function S such that for each element y_i of Y, either $\sum_{x \in S^{-1}(y_i)} f_x \leq \frac{2}{256}$ or $S^{-1}(y_i)$ is a single element of X.*

The theorem and corollary prove that we can create mappings that limit the amount of values in the base data assigned to any non unique code. This directly implies that we can limit the amount of times we need to access the base data. We note that Theorem 5.3.1 and Corollary 1 apply when the domain X is a compound space. For instance, X could be the domain of (country, city, biological sex, marital status, employment status) and the theorem would still apply. Appendix C.1 includes further discussion and experiments with multi-column Column Sketches.



(a) Uniform Distribution



(b) Normal Distribution

Figure 5.3: Bucket sizes track the distribution of the data.

5.3.3 Numerical Compression Maps

Lossless Numerical Compression. For numeric data types, lossless compression techniques such as frame-of-reference (FOR), prefix suppression, and null suppression work by storing the value as a relative value from some offset [Zukowski et al., 2005, Westmann et al., 2000, Fang et al., 2010]. All three techniques support operations in compressed form; in particular, they can execute equality predicates, range predicates, and aggregation operators without decompressing. However, to support aggregation efficiently, each of these techniques conserves differences; that is, given base values a and b, their encoded values e_a and e_b satisfy $e_a - e_b = a - b$. This limits their ability to change the entropy of high order bits, and these bits can only be truncated if every value in a column has all 0's or all 1's on these high order bits.

Constructing Numerical Compression Maps. In contrast to lossless techniques, lossy compression is focused only on maximizing the utility of the bits in the sketch. The simplest way to do this while preserving order is to construct an equi-depth histogram which approximates the CDF of the input data, and then to create codes based on the endpoints of each histogram bucket. When given a value in our numerical domain, the output of the map is then simply the histogram bucket that a value belongs to. We create approximately equi-depth histograms by sampling values uniformly from the base column, sorting these values, and then generating the endpoints of each bucket based off of this sorted list.

Because histogram buckets are contiguous, storing the endpoint of each bucket i is enough to know the range that histogram bucket covers. Figures 5.3a and 5.3b show examples of mappings using histograms for two different data sets. In both graphs, we use 200,000 samples to create 256 endpoints. The uniform distribution goes from 0 to 10,000,000 and the normally distributed data is of mean 0 and variance 1000. The codes for the uniform distribution are evenly spaced throughout, and the codes for the normal distribution are farther apart towards the endpoints of the distribution and closer together towards the mean. The histograms capture the distributions of both functions and evenly space the values in the base data across the codes.

Handling Frequent Values. We define a frequent value to be a value that appears in more than $\frac{1}{z}$ of the base data values. To handle these frequent values, we first perform the same procedure as before and create a sorted list of sampled values. If a value represents more than $\frac{1}{z}$ of a sample of size n, then one of the values in the sorted list at $(\frac{n}{z}, \frac{2n}{z}, \ldots, \frac{(z-1)n}{z})$ must be that value. Thus, for each of these z values we can search for its first and last occurrence to check if it represents more than $\frac{1}{z}$ of the sample. If so, mark the middle position of that value in the list and give the value the unique code $c * \frac{midpoint}{n}$ (rounded to nearest integer), where c is the number of codes in the Column Sketch. In the case that $z < c$ and that two values would be given the same unique code c, the more frequent value is given that unique code. In this paper, we use $z = 256$. Though a smaller value of z can create faster average query times, we chose 256 so that making a code unique does not increase the proportion of values in non-unique codes.

After finding the values that deserve a unique code and giving them associated code values, we equally partition the sorted lists between each unique code and assign the remaining code values accordingly. The identification of unique codes is in the worst case comparable to a single pass over the sample, and the partitioning of non-unique codes is then a constant time operation.

To make it so that updates cannot force a re-encoding, we do not allow unique codes to occupy subsequent positions. If in the prior procedure values v_i and v_{i+1} would be given unique code i and $i+1$ respectively, only the more frequent value is given a unique code. For values to be assigned subsequent codes, the less frequent code can contain no more than $\frac{3}{2c}$ of the sampled values, and so our previous robustness results for no non-unique code having too many values still hold.

Additionally, we do not allow the first and last codes in the compression map to be unique.

Estimating the Base Data Distribution. For the compression map to have approximately equal numbers of values in each code, the sampled histogram created from the empirical CDF needs to closely follow the distribution of the base data. The Dvoretzky-Kiefer-Wolfowitz inequality provides bounds on the convergence of the empirical CDF F_n of n samples towards the true CDF F, stating: $P(\sup_{x \in \mathbb{R}} \|F_n(x) - F(x)\| \geq \epsilon) \leq 2e^{-n\epsilon^2}$ [Massart, 1990, Dvoretzky et al., 1956]. In this equation we can treat F, the true distribution, as an unknown quantity and the column as an i.i.d. sample of F, or we can treat the column as a discrete distribution having CDF exactly equal to the CDF of the base data. In both cases, sampling from the base data n times gives the required result on the distance of our sampled data's empirical CDF F_n from the true CDF F[1]. We prove in Section 5.7 that any column with less than $\frac{4}{256}$ of the base data provides $2\times$ performance benefits during scans, and a Column Sketch mapping never assigns a single non-unique code any proportion of values that it estimates as over $\frac{2}{256}$. Therefore, we targeted $\epsilon = \frac{2}{256}$. With 200,000 samples as in Figure 5.3, the chance of an error of this amount is less than 10^{-5}. Both the number of samples n and the desired ϵ are tunable.

In Appendix C.5, we provide ways to deal with columns whose value distribution shifts over time.

5.3.4 Categorical Compression Maps

Categorical Data and Dictionary Encoding. Unlike numerical distributions, categorical distributions often have values which take up significant portions of the dataset. Furthermore, certain categorical distributions have no need for ordering.

Traditionally, categorical distributions have been encoded using (optionally order preserving) fixed width dictionary encoding. Dictionary encoding works by giving each unique value its own numerical code. A simple example is the states in the United States. While this might be declared as a varchar column, there will be only 50 distinct values and so each state can be represented by a

[1]Under the case that our column is a considered an i.i.d. sample from F, this should be without replacement. For the case that our columns CDF is our desired CDF, this should be with replacement.

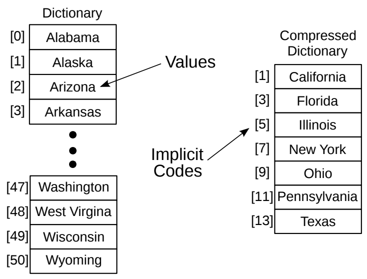


Figure 5.4: Lossy dictionaries trade uniqueness for better compression.

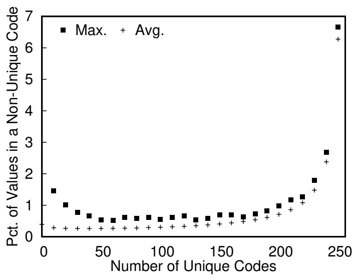


Figure 5.5: The number of unique codes should be neither too high nor too low.

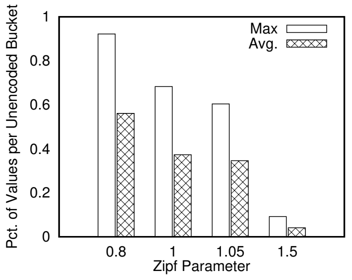


Figure 5.6: Regardless of the level of skew, the number of values in non-unique codes is low.

number between 0 and 49. Since each distinct value requires a distinct code, the number of bits needed to store a dictionary encoded value is $\lceil \log_2 n \rceil$, where n is the number of unique values.

Lossy Dictionaries. The compression maps for categorical distributions look similar to dictionary encoding, except that rare codes have been collapsed in on each other, making the number of code values smaller. The primary benefit of this collapsing is that a scan of the sketched column reads less memory. However, there is also a processing benefit as we can choose the number of code values in a non-injective encoding so that codes are of fixed byte length. For instance, if we look at a dataset with 1200 unique values, then a dictionary encoded column needs 11 bits per value. If these codes are packed densely one after the other, they will not begin at byte boundaries and the CPU will need to unpack the codes to align them on byte boundaries. If they are not packed densely, then the codes are padded to 16 bits, which in turn brings higher data movement costs. With Column Sketches, the lossy encoding scheme can choose the number of bits to be a multiple of 8, saving data movement without creating the need for code unpacking.

Shown in Figure 5.4 is an example comparing an order-preserving dictionary to an order-preserving lossy dictionary for states in the United States. Only the unique codes are shown, with the non-unique codes in the implied gaps. Though this is a simplified example, it shows various properties that we wish to hold for lossy dictionaries. The most frequent values, in this case the most populous states, are given unique codes, whereas rarer values share codes. California is given the unique code 1 whereas Wyoming shares code 14 with Wisconsin, West Virginia and several other states. The 7 unique codes cover nearly 50% of the population of the U.S. The other 50% of the population are divided amongst the 8 non-unique codes, with each non-unique codes having an expected 6.25% of the data. However, this is due to the low number of non-unique codes. For instance, if we were to change this to cities in the United States, of which there are around 19,000, and have 128 unique codes and 128 non-unique codes, then each non-unique code would have only 0.6% of the data in expectation.

Unordered Categorical Data. We first discuss assigning codes for categorical distributions that have no need for data ordering. Without the need for ordered code values, we are free to assign any value to any code value. This freedom of choice makes the space of possible compression maps very large, but also gives rise to fairly good intuitive solutions. We have three major design decisions:

1. How many values should be given unique codes?
2. Which values do we give unique codes?
3. How do we distribute values amongst the non-unique codes?

(1) Assigning Unique Codes. The simplest approach to assigning unique codes is to give the most frequent values unique codes. This is robust in that it bounds the amount of times we access the base data for any predicate. More aggressive (but potentially less robust) approaches which analyze query history to assign unique code values are presented in Appendix C.4.

(2) Number of Unique Codes. The choice of how many unique codes to create is a tunable design decision depending on the requirements of the application at hand. We describe here two ways to make this decision. One way is to give every value that occurs with more than some frequency z in our sample a unique code value, leaving the remaining codes to be distributed amongst all values with frequency less than the specified cutoff. This parameter z has the same tradeoffs as in the ordered case, and tuning it to workload and application requirements is part of future work. In this paper, z is set to 256, for analogous reasons to the ordered case. The second way of assigning unique codes is to set a constant value for the number of codes that are unique. The second method works particularly well for certain values. For instance, if exactly half the assigned codes are unique codes, then we can use the first or last bit of code values to delineate unique and non-unique codes.

(3) Assigning Values to Non-Unique Codes. The fastest method for ingesting data is to use a hash function to relatively evenly distribute the values amongst the non-unique codes. If there are c codes and u unique codes, we assign the unique codes to codes $0, 1, \dots, u-1$. When encoding an incoming value, we first check a hash table containing the frequent values to see if the incoming value is uniquely encoded. If the value is uniquely encoded, its code is written to the sketched column. If not, the value is then encoded as $u + [h(x)\%(c-u)]$.

Analysis of Design Choices. By far the most important characteristic for performance is making sure that the most frequent values are given unique codes. Figures 5.5 and 5.6 show the maximum number of data items given to any non-unique code as well as the average across all non-unique codes. In both figures, we have 100,000 tuples given 10,000 unique values, with the frequency with which we see each value following a Zipfian distribution. The rare values are distributed amongst the non-unique codes by hashing. In the first figure, we keep the skew parameter at 1 and vary the number of unique codes. In the second graph, we use 128 unique codes and change the skew of the dataset. As seen in Figure 5.5, choosing a moderate number of unique codes guarantees each non-unique code has a reasonable number of values in the base data. Figure 5.6 shows that for datasets with both high and low skew, the number of tuples in each non-unique code is a small proportion of the data.

Ordered Categorical Data. Ordered categorical data shares properties of both unordered categorical data and of numerical data. Like numerical data, we expect to see queries that ask questions about some range of elements in the domain. Like unordered categorical data, we expect to see queries that predicate on equality comparisons. Spreading values in the domain evenly across codes achieves the properties needed by both. Thus the algorithm given for identifying frequent values in numerical data works well for ordered categorical data as well.

5.4 Predicate Evaluation Over Column Sketches

For any predicate that a Column Sketch evaluates, we have codes which can be considered the endpoint of the query. For instance, the comparison $B < x$ given as an example in Section 5.2 has the endpoint $S(x)$. For range predicates with both a less than and greater than clause, such as $x_1 < B < x_2$, the predicate has two endpoints: $S(x_1)$ and $S(x_2)$. And while technically an equality

Algorithm 6 Column Sketches Scan
Select where $B < x$, Column Sketch of one byte values

```
Input: S is a function which is order preserving
 1: repeat_sx = _mm128_set1_epi8($S(x)$)
 2: for each segment $b$ in sketched column do
 3:                                   Work over the code one block of data at a time
 4:   position= $b\times$ segment size
 5:   for number of simd iterations in segment size do
 6:       Perform logical comparisons necessary for predicate evaluation on each code. The 1,2 denote that this is
        done twice
 7:     codes1,2 = _mm_load_si128(codes_address)
 8:     definite1,2 = _mm_cmplt_epi8(codes,repeat_sx)
 9:     possible1,2 = _mm_cmpeq_epi8(codes,repeat_sx)
10:     bitvector_def1,2 = _mm128_movemask_epi8(definite1)
11:     bitvector_possible1,2 = _mm128_movemask_epi8(possible1)
12:                                                    Store results we are certain of
13:     bitvector_def = (bitvector_def1 « 16) | bitvector_def2
14:     store(bitvector_def) into result
15:            Check if the boundary values have any matching tuples and store qualifying positions.
16:     conditional store bitvector_possible1,2 into temp_result.
17:     position + = 32
18:                                           Check all tuples that we are uncertain about
19:   for position in temp_result do
20:     if B[position] == x then
21:       set bit position + beginning position in result
```

predicate has no endpoint since it isn't a range, for notational consistency we can think of $S(x)$ as an endpoint of the predicate $B = x$.

SIMD Instructions. SIMD instructions provide a way to achieve data-level parallelism by executing one instruction over multiple data elements at a time. The instructions look like traditional CPU instructions such as addition or multiplication, but have two additional parameters. The first is the size of the SIMD register in question and is either 64, 128, 256, or 512 bits. The second parameter is the size of the data elements being operated on, and is either 8,16,32, or 64. For example, the instruction _mm256_add_epi8 (___m256i a, ___m256i b) takes two arrays, each with 32 elements of size 8 bits, and produces an array of thirty-two 8 bit elements by adding up the corresponding positions in the input in one go.

Scan API. A Column Sketch scan takes in the Column Sketch, the predicate operation, and the values of its endpoints. It can output a bitvector of matching positions or a list of matching positions, with the default output being a bitvector. In general, for very low selectivities a position list should be used and for higher selectivities a bitvector should be used. This is because at high selectivities the position list format requires large amounts of memory movement.

Scan Procedure. Algorithm 2 depicts the SIMD based Column Sketch scan using Intel's AVX instruction set and producing bitvector output. For space reasons, we omit the setup of several variables and use logical descriptions instead of physical instructions for lengthier operations.

The inner part of the nested loop is responsible for the logical computation of which positions match and which positions possibly match. In the first line, we load the 16 codes we need before performing the two logical comparisons we need. For the less than case, our only endpoint is $S(x)$, and we check for this value using the equality predicate on line 10. For each position matching this predicate, we will need to go to the base data.

After these comparisons, we translate the definitely qualifying positions into a bitvector and store these immediately. For the possibly matching positions, we perform a conditional store. Left out of our code for reasons of brevity, the conditional store first checks if its result bitvector is all zeros. If it is not, it translates the conditional bitvector into a position list and stores the results in a small buffer on the stack. The result bitvector for possibly matching values will usually be all zeros as the Column Sketch is created so that no code holds too many values, and so the code to translate the bitvector into a position list and store the positions is executed infrequently. As a small detail, we found it important that the temporary results be stored on the stack. Storing these temporary results on the heap instead was found to have a 15% performance penalty.

The Column Sketch scan is divided into a nested loop over smaller segments so the algorithm can patch the result bitvector using the base data while the result bitvector remains in the high levels of CPU cache. If we check the possibly matching positions all at the end, we see a minor performance degradation of around 5%.

Unique Endpoints. Unique endpoints make the scans more computationally efficient. If the code $S(x)$ is unique, there is no need to keep track of positions and no need for conditional store instructions. Furthermore, the algorithm only needs a single less than comparison. After that comparison, it immediately writes out the bitvector. More generally, given a unique code, a scan over a Column Sketch completely answers the query without referring to the base data, and thus looks exactly like a normal scan but with less data movement.

Equality and Between Predicates. Equality predicates and between predicates are processed similarly to Algorithm 2. For equality predicates, the major difference is that, depending on whether the code is unique, the initial comparison only needs to store the partial result or the definite result. It can drop the other store instruction. For two-sided ranges with two-non unique endpoints, we have both $>$ and $<$ comparisons and an equality check on both endpoints. The list of possible matches is the logical or of the two equality checks, and the list of definite matches is the logical and of the two inequality comparisons. The rest of the algorithm is identical. If an endpoint is unique, then similar to the one sided case, the equality comparison for that endpoint can be removed.

Two Byte Column Sketches. Previous descriptions are based on single byte Column Sketches. In case we have a two byte representation, the logical steps of the algorithm remain the same. The only change is replacing the 8 bit SIMD banks with 16 bit SIMD banks.

5.5 Performance Modeling

The performance model for Column Sketches assumes that performance depends on data movement costs. This assumption is justified in our experiments for byte-aligned Column Sketches, where we show that a Column Sketch scan saturates memory bandwidth.

Notation. Let B_b be the size of each value in the base data in bytes and let B_s be the size of the codes used in the sketch (both possibly non-integer such as 7/8 for a 7 bit Column Sketch). Let n be the total number of values in the column, and let M_g be the granularity of memory access. Because the modeling in this section is aimed at main memory, we use $M_g = 64$. The analysis of stable storage based systems is given in Appendix C.2.

Model: Bytes Touched per Value. Let us assume that the Column Sketch has no unique codes and consider a cache line of data in the base data. This cache line is needed by the processor if at least one of the corresponding codes in the sketched column matches the endpoint of the query. If we assume that there is only one endpoint of the query, then the probability that any value takes on the endpoint code is $\frac{1}{2^{8B_s}}$. Therefore, the probability that no value in the cache line takes on the endpoint code is approximately $1 - (\frac{1}{2^{8B_s}})^{\lceil \frac{M_g}{B_b} \rceil}$, with the ceiling coming from values which have part of their data in the cache line. The chance we touch the cache line is the complement of that number, and so the total number of bytes touched per value is

$$B_s + B_b[1 - (1 - \frac{1}{2^{8B_s}})^{\lceil \frac{M_g}{B_b} \rceil}] \tag{5.1}$$

Plugging in 4 for B_b, 1 for B_s, and 64 for M_g, we get the value 1.24 bytes. If we use 8 for B_b, this remains at 1.24 bytes. If we change this to a query with two endpoints, the $\frac{1}{2^{*B_s}}$ term becomes $\frac{2}{2^{*B_s}}$ and so the equation becomes $B_s + B_b[1 - (1 - \frac{2}{2^{8B_s}})^{\lceil \frac{M_g}{B_b} \rceil}]$ From here, if we keep $B_s = 1$ and $M_g = 64$, then $B_b = 4$ gives an estimated cost of 1.47 bytes. Again, using $B_b = 8$ gives 1.47 bytes as well. Thus, for both one and two endpoint queries, and for both 4 byte and 8 byte base columns, a Column Sketch scan has significantly less data movement than a basic table scan.

We now take into account unique codes. Assume we follow the technique for deciding on unique codes from Section 5.3.3, where unique codes are given to values that take more than $\frac{1}{256}$ of the sample. Since the codes partition the dataset, the non-unique codes contain less than $\frac{1}{256}$ of the dataset on average. Following similar logic to above, the result is that creating unique codes decreases the expected cost of a Column Sketch scan in terms of bytes touched for non-unique codes. For unique codes the number of bytes touched per value is 1. More detail is given on how unique codes affect the model in Appendix C.3.

Performance Tradeoffs in Code Size. Assume for the moment that non-byte aligned scans are memory-bound. Equation (1) gives a simple optimization problem which gives to the approximate optimal # of bits per code for scans, with more bits per code increasing the amount of memory bandwidth required to perform the scan but decreasing the needed number of base data accesses. After optimizing this value though, we have various tradeoffs. First and most apparent, more bits per code creates a larger memory footprint for the Column Sketch. Second, more bits per code means a larger dictionary, which slows down ingestion for ordered Column Sketches as more comparisons are needed. Finally, more bits per code makes the performance of the Column Sketch scan more robust. This is because the left side of equation (1) is constant, whereas the right side is variable ($\frac{1}{2^{8B_s}}$ was the expected proportion of codes for an endpoint). By increasing the number of bits per code, we reduce the influence of the right side of the cost equation and therefore reduce the variance of equation (1). Currently, we find that equation (1) only holds for byte-aligned code sizes with the scan performance being worse by around 30% for non-byte aligned codes, and so the tradeoffs mostly favor $B_s = 1$ or $B_s = 2$.

5.6 System Integration and Memory Overhead

System Integration. Many components of Column Sketches already exist partially or completely in mature database systems. Creating the compression map requires sampling and histograms, which are supported in nearly every major system. The SIMD scan given in Section 5.4 is similar to optimized scans which already exist in analytic databases, and Zone Maps over the base data can filter out the corresponding positionally aligned sections of a Column Sketch. Adding data and updating data in a Column Sketch are very similar to data modifications in columns which are dictionary encoded. In Section 5.7, we show that a Column Sketch scan is always faster than a traditional scan. Thus, optimization can use the same selectivity based access path selection between traditional indices and the Column Sketch, with a lower switch point. As well, Column Sketches work naturally over any ordered data type that supports comparisons. This contrasts with related techniques such as early pruning techniques, which need modifications to various types such as floating point numbers to make them binary comparable. Finally, Column Sketches makes no change to the base data layout and so all other operators except for select can be left unchanged.

Memory Overhead. Let b_s be the number of bits per element in the Column Sketch. Then we need $b_s \times n$ bits of space for the sketched column. If we let b_b be the number of bits needed for a base data element, then each dictionary entry needs $b_b + 1$ bits of space, where the extra bit comes from marking whether the value for that code is unique. The size of the full dictionary is then $(b_b + 1) \times 2^b$ bits. Notably, b is usually quite small (we use $b = 8$ at all points in this paper to create byte alignment) and so the dictionary is also usually quite small. Additionally, the size of the dictionary is independent of n, the size of the column, and so the overhead of the Column Sketch approaches $b_s \times n$ bits as n grows. Additionally, we note that a Column Sketch works best with compression techniques on the base column that allow efficient positional access. This is normally the case for most analytical systems when data is in memory, as data is usually compressed using fixed width encodings [Abadi et al., 2013]. For a discussion of disk-based systems, we point the reader to Appendix C.2.

5.7 Experimental Analysis

We now demonstrate that, contrary to state of the art predicate evaluation methods, Column Sketches provide an efficient and robust access method regardless of data distribution, data clustering, or selectivity. We also show that Column Sketches efficiently ingest new data of all types, with order of magnitude speedups for categorical domains.

Competitors. We compare Column Sketches against an optimized sequential scan, BitWeaving/V (noted from here on out as just BitWeaving), Column Imprints and a B-tree index. The scan, termed FScan, is an optimized scan over numerical data which utilizes SIMD, multi-core, and zone-maps. For BitWeaving and Column Imprints, we use the original code of the authors [Li and Patel, 2013, Sidirourgos and Kersten, 2013], with some minor modifications to Column Imprints to adapt it to the AVX instruction set. We additionally compared against a SIMD version of BitWeaving [Polychroniou et al., 2015], but found the SIMD version performed slightly less efficiently. The B-tree utilizes multi-core and has a fanout which is tuned specifically for the underlying hardware. In addition, for categorical data we compare Column Sketches against BitWeaving and "SIMD-Scan" from [Willhalm et al., 2013, 2009], which is a SIMD scan that operates directly over bit-packed dictionary compressed data. Since we use SIMD at various points that are not referring to "SIMD-Scan", we refer to this technique as CScan. All experiments are in-memory; they include no disk I/O.

Scan API. The outputs of the scan procedure for a Column Sketch, BitWeaving, Column Imprints and FScan are identical. As an input the scan takes a single column and as an output it produces a single bitvector. The B-tree index scan takes as input a single column and outputs a list of matching positions sorted by position. This is because B-trees store their leaves as position lists and so this optimizes the B-tree performance.

Infrastructure. We run our experiments on a machine with 4 sockets, each equipped with an Intel Xeon E7-4820 v2 Ivy Bridge processor running at 2.0GHz with 16MB of L3 cache. Each processor has 8 cores and supports hyper-threading for a total of 64 hardware threads. The machine includes 1TB of main memory distributed evenly across the sockets and four 300GB 15K RPM

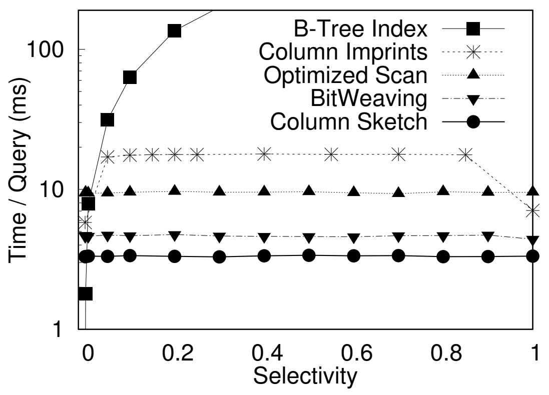


Figure 5.7: Column Sketches outperform other access methods in all but the most selective queries.

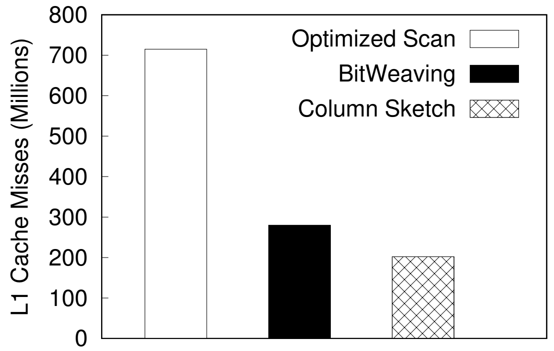


Figure 5.8: Column Sketches incur fewer cache misses than competing techniques.

disks configured in a RAID-5 array. We run 64-bit Debian "Wheezy" version 7.7 on Linux 3.18.11. To eliminate the effects of NUMA on performance, each of the experiments is run on a single socket. We give performance measurements in terms of cycles per tuple. For this machine and using a single socket, a completely memory bound process achieves a maximum possible performance of 0.047 cycles per byte touched by the processor.

Experimental Setup. Unless otherwise noted, the column used consists of 100 million values. When conducting predicate evaluation, each method is given use of all 8 cores. The numbers reported are the average performance across 100 experimental runs.

5.7.1 Uniform Numerical Data

Fast and Robust Data Scans. Our first experiment demonstrates that Column Sketches provide efficient performance regardless of selectivity. We test over numerical data of element size four bytes, distributed uniformly throughout the domain, and we vary selectivity from 0 to 1. The predicate is a single sided $<$ comparison, with the endpoint of the query being a non-unique code of the Column Sketch. For this experiment only, we report performance as milliseconds per query, as the metric cycles/tuple is not very informative for the B-Tree.

Figure 5.7 shows the results. It depicts the response time for all five access methods as we vary selectivity. For very low selectivities below 1%, the B-tree outperforms all other techniques.

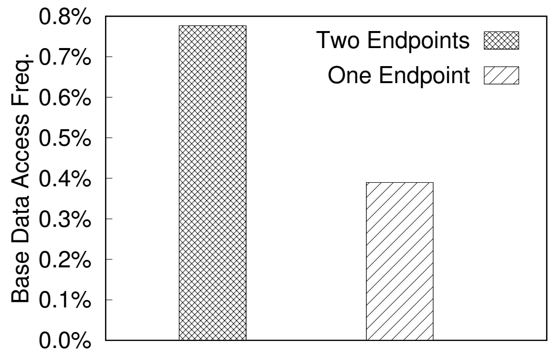


Figure 5.9: Column Sketches need to access the base data very infrequently.

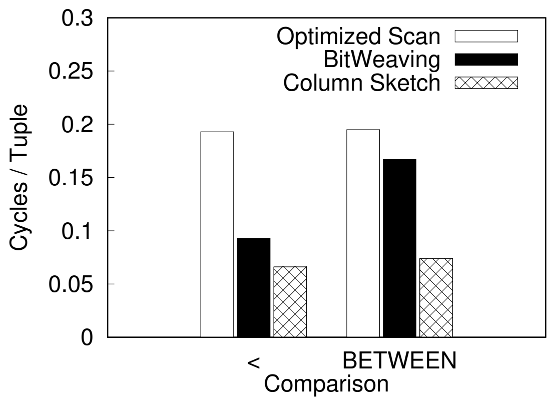


Figure 5.10: Column Sketches need fewer cycles to evaluate single and double sided predicates.

However, once selectivity reaches even moderate levels, the B-tree performance significantly degrades. This is because the B-tree index needs to traverse the leaves at the bottom level of the tree, which stalls the processor and so wastes cycles. In contrast to the B-tree, the Column Sketch, Column Imprints, BitWeaving, and FScan view data in the order of the base data and consistently feed data to the processor.

During predicate evaluation, BitWeaving, Column Imprints, the Column Sketch, and the optimized scan all continually saturate memory bandwidth. However, the Column Sketch performs the best by reading the fewest number of bytes, outperforming the optimized scan by 2.92×, Column Imprints by $1.8 - 4.8\times$ and BitWeaving by 1.4×. Figure 5.8 breaks down these results, showing the number of L1 Cache Misses for the Column Sketch against its two closest competitors, FScan and BitWeaving. The results align closely with the performance numbers, with the Column Sketch seeing 1.39× fewer L1 cache misses than BitWeaving and 3.54× fewer L1 cache misses than the optimized scan.

In performing predicate evaluation, the optimized scan and Column Imprints see nearly every value, leading to their high data movement costs. This is because the Zone Map and Column Imprint work best over data which is clustered; when data isn't clustered, as is the case here, these techniques provide no performance benefit. BitWeaving and the Column Sketch also see every value, but decrease data movement by viewing fewer bytes per value. BitWeaving achieves this via early pruning, but this early pruning tends to start around the 12th bit. Additionally, even if BitWeaving eliminates all but one item from a segment by the 12th bit, it may need to fetch that group for comparison multiple times to compare the 13th, 14th, and so on bits until the final item has been successfully evaluated. In contrast to BitWeaving, the Column Sketch prunes most data by the time the first byte has been observed, with Figure 5.9 showing that a single endpoint query accesses around 0.4% of tuples in the base data. As well, in the rare case an item hasn't been resolved by the first byte, the Column Sketch goes directly to the base data to evaluate the item. This makes sense, once early pruning has reached a sparse stage where most tuples have or have not qualified, query execution should directly evaluate the small number of values left over.

Figure 5.10 shows the performance of the Column Sketch, BitWeaving, and the optimized scan in terms of cycles/tuple across single comparison and between predicates. In both cases, the Column Sketch performs significantly better than FScan and BitWeaving. For Fscan, its performance across both types of predicates is completely memory bandwidth bound and constant at 0.195 cycles/tuple. For BitWeaving, its performance on the between predicate is nearly half of it single comparison performance, going from 0.093 cycles/tuple to 0.167. However, this is a side effect of the distribution code we were given, which evaluates the $<$ predicate completely before evaluating the $>$ predicate after. If the two predicates were evaluated together, we would expect to see only a small decrease in performance. For the Column Sketch, it sees a minor drop in performance from 0.066 cycles/tuple to 0.074 cycles/tuple. This small drop in performance of the Column Sketch comes from having two non-unique endpoints, and not from increased computational costs. In the case one of the two endpoints is unique, the performance stays at 0.066 cycles/tuple. If both endpoints are unique, the performance is 0.053 cycles/tuple.

Model Verification. The performance of the Column Sketch nearly exactly matches what our model predicts. The model predicts that we would touch 1.24 bytes for a single non-unique endpoint, and 1.47 bytes for two endpoints. Taking into account the result bitvector and multiplying this by our saturated bandwidth performance of 0.047 cycles/byte, we get an expected performance from our model of 0.064 for a single endpoint and 0.075 for two endpoints.

Robustness to a Bad Compression Map. Figure 5.11 shows the performance of the Column Sketch as more and more data is put into the single non-unique endpoint of our $<$ comparison. The

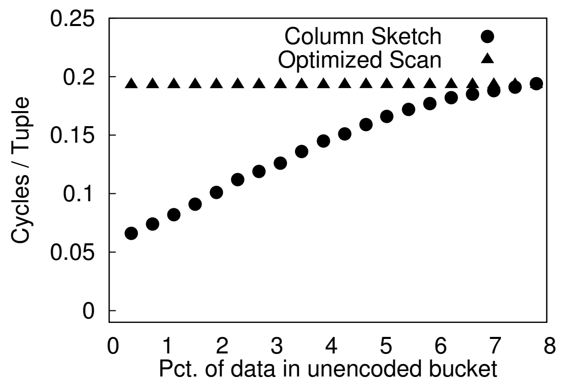


Figure 5.11: Column Sketches perform well even in the case of a bad compression map.

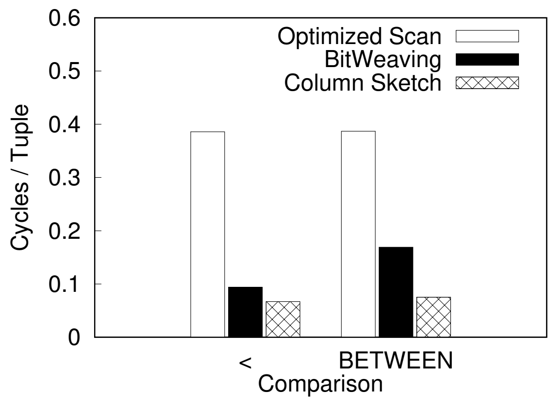
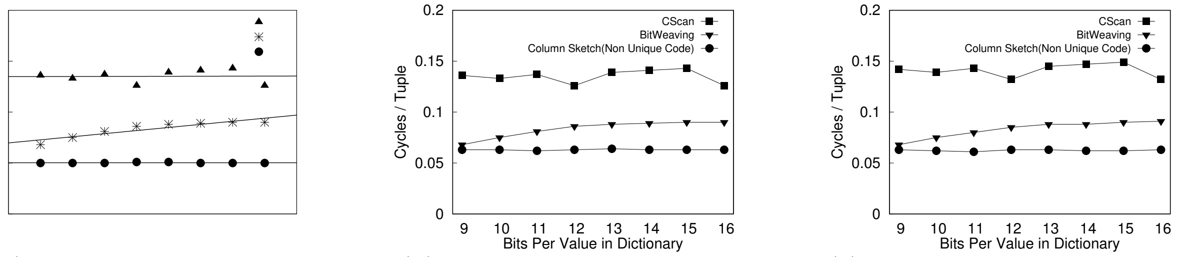


Figure 5.12: Column Sketches retain nearly identical performance with larger base data.



(a) Equality, Unique Code **(b)** Equality, Non-Unique Code **(c)** Less Than, Non-Unique Code

Figure 5.13: Column Sketches is the fastest scan across unique and non-unique codes, and across equality and range comparisons

leftmost data point in the graph shows the expected performance of the Column Sketch, i.e. when the non-unique code has $\frac{1}{256}$ of the data, and each subsequent data point has an additional $\frac{1}{256}$ of the data assigned to that code. Notably, our analysis from Section 5.3.3 says that data points beyond the first 4 points will occur with probability less than $\frac{1}{10^5}$. The eventual crossover point when the performance of a Column Sketch degrades to worse than the performance of a basic scan is when the single endpoint holds $\frac{20}{256}$ of the data.

Larger Element Sizes. Our second experiment shows how performance changes for traditional scans, BitWeaving, and Column Sketch scans as the element size increases from four to eight bytes. The setup of the experiment is the same as before, i.e., we observe the response time for queries over uniformly distributed data. We do not use the index or Column Imprints from now on, as across all experiments the two closest competitors are FScan and BitWeaving. The results are shown in Figure 5.12.

Beta	BitWeaving (cycles/code)	Column Sketch (cycles/code)
1 (Uniform)	0.092	0.066
5	0.112	0.066
10	0.118	0.066
50	0.139	0.066
500	0.152	0.066
5000	0.188	0.066

Table 5.1: Scan Performance under Skewed Datasets

For FScan, the larger element size means a proportional decrease in scan performance, with codes/cycle going from 0.193 to 0.386. However, for a Column Sketch, we can control the given code size independently of the element size in the base data. Thus, since we aim the Column Sketch at data in memory, we keep the code size at one byte. The scan is then evaluated nearly identically to the scan with base element size four bytes, and has nearly identical performance (0.067 instead of 0.066 cycles/code). Similarly, BitWeaving prunes almost all data by the 16th bit and so sees a negligible performance increase of 0.01 cycles/code. The overall performance increase from using the Column Sketch is 5.76× over the sequential scan and 1.4× over BitWeaving.

5.7.2 Skewed Numerical Data

Experiment Setup. For skewed data we use the Beta distribution scaled by the maximum value in the domain. The Beta distribution, parameterized by α and β is uniform with $\alpha = \beta = 1$ and becomes more skewed toward lower values as β increases with respect to α. We use this instead of the more commonly seen zipfian distribution as the zipfian distribution is more readily applied to categorical data with heavy skew, whereas the Beta distribution is continuous and better captures numerical skew. In the experiment, the element size is 4 bytes. We keep α at 1, and vary the β parameter. All queries are a range scan for values less than $2^8 - 1 = 255$. Table 2 depicts the results (FScan is not included as its performance is identical to Figure 5.7).

Robustness to Data Distribution. As the distribution gets more skewed towards low values, the high order bits tend to be mostly 0s. For BitWeaving, this means the high order bits don't give much data pruning, and as a result the performance of BitWeaving tends to look more like the full column scan. This is a weakness of using encoding which preserves differences, if there are even a trace amount of high values, the high order bits must be nearly all zero. For BitWeaving, we see performance degradation fairly quickly, as even $\beta = 5$ produces a notable performance drop. As β increases, the performance degrades more. In contrast, the performance of the Column Sketch is stable, with the ability to prune data unaffected by the data distribution change.

5.7.3 Categorical Data

Categorical Attributes. Our second set of experiments verifies that Column Sketches provide performance benefits over categorical attributes as well as numerical attributes. Unlike the numerical attributes seen in the previous experiments, categorical data consists of a significant number of frequent items. The Column Sketch was encoded with values taking up more than $\frac{1}{256}$ of the data being given unique codes. The resulting data set had 65 unique codes and 191 non-unique codes, with unique codes accounting for 50% of the data and non-unique codes accounting for 50% of the data.

For the dictionary compressed columns, we vary the number of unique elements so that the value size in the base data is between 9 and 16 bits. This matches the size of dictionary compressed columns which take up the majority of execution time in industrial workloads [Willhalm et al., 2009]. For columns whose base data takes less than 9 bits, Column Sketches provide no benefit and systems engineers should use other techniques.

In Figure 5.13a, we see that using a Column Sketch to perform equality predicate evaluation is faster than BitWeaving and CScan. Surprisingly, the improvement is more more than a 12% improvement over BitWeaving, even for 9 bit values, and sees considerably more improvement against CScan. For Column Sketches, the performance improvement increases up to bit 12 against BitWeaving, at which point the performance of BitWeaving becomes essentially constant. For the CScan, the improvement varies based off the base column element size. Due to word alignment, CScan does better on element sizes that roughly or completely align with words (such as 12 and 16 bits). However, for all element sizes, the computational overhead of unpacking the codes and aligning them with SIMD registers makes it so CScan is never completely bandwidth bound.

For non-unique codes, the performance of the Column Sketch is only slightly worse. The performance drops by 15% as compared to unique codes, with the Column Sketch remaining more performant than BitWeaving and CScan across all base element sizes. This is again due to the very limited frequency with which the Column Sketch looks at the base data; in this case, the Column Sketch is expected to view the base data for one out of every 256 values.

We additionally conduct range comparisons on categorical data, as shown in Figure 5.13c. The results are similar to Figure 5.13b. For Column Sketches we observe similar performance both when we stored the the base data as a text blob and when it was stored using non order-preserving dictionary encoded values. This is notable, since both blob text formats and non-order preserving dictionary encoded values are easier to maintain during periods in which new domain values appear frequently.

5.7.4 Load Performance

Experiment Setup. We test load performance for both numerical and categorical data, showing that Column Sketches achieve fast data ingestion regardless of data type. For both data ingestion experiments, we load 100 million elements in five successive runs. Thus at the end, the table has 500 million elements. In the numerical experiment, the elements are 32 bits and the Column Sketch contains single byte codes. In the categorical experiment, the elements are originally strings. For BitWeaving, we turn these strings into 15 bit order preserving dictionary encoded values, so that the BitWeaved column can efficiently conduct range predicates. For the Column Sketch, we encode the strings in the base data as non-order preserving dictionary encoded values, and use an order-preserving Column Sketch. As shown in the prior section, this is efficient at evaluating range queries. The time taken to perform the dictionary encoding for the base data is not counted for BitWeaving or Column Sketches. The time taken to perform encoding for the Column Sketch is counted.

The categorical ingestion experiment is then run under two different settings: in the first setting, each of the five successive runs sees some new element values, and so elements can have their encoded values change from run to run. Because there are new values, the order preserving dictionary needs to re-encode old values, and so previous values may need to be updated. In the second setting,

Technique	Numerical Data	Categorical Data(No New Values)	Categorical Data(New Values)
BitWeaving	50.823	26.139	80.155
Column Sketches	5.483	5.379	5.531

Table 5.2: Time to load all batches (sec.)

there are no new values after the first batch.

Fast Ingestion via Column Sketches. Column Sketches tend to have fast load performance as the only transformation needed on each item is a dictionary lookup. The data can then be written out as contiguous byte aligned codes. As well, regardless of the new values in each run, the Column Sketch always has a non-unique code for each value and thus never needs to re-encode its code values. Thus, a Column Sketch is particularly well suited to domains that see new values frequently, with the Column Sketch allowing for efficient range scans without requiring the upkeep of a sorted dictionary. In contrast to Column Sketches, BitWeaving tends to have a high number of CPU operations to mask out each bit and writes to scattered locations. More importantly, new element values can cause prior elements to need to be re-encoded. Notably, this isn't particular to BitWeaving, but an inherent flaw in any lossless order-preserving dictionary encoding scheme, with the only solution to include a large number of holes in the encoding [Binnig et al., 2009].

5.8 Related Work

Compression and Optimized Scans. The tight integration of compression and execution into scans in column-oriented databases started in the mid-2000's with MonetDB and C-Store [Zukowski et al., 2005, Abadi et al., 2006, Zukowski et al., 2005]. Since then work has been done integrating numerous types of compression into scans, notably dictionary compression, delta encoding, frame-of-reference encoding, and run-length encoding [Graefe and Shapiro, 1991, Zukowski et al., 2006]. Nowadays, mixing compression and execution is standard and is seen in most commercial DBMSs [Raman et al., 2008, Barber et al., 2012, Färber et al., 2012, Lamb et al., 2012]. Recently, IBM created Frequency Compression [Raman et al., 2008, 2013], exploiting data reordering for better scan performance.

Each of these techniques is lossless and designed to be used for base data. Thus, these techniques could achieve higher compression ratios through the use of lossy instead of lossless compression, reducing memory and disk bandwidth during scans. The potential of lossy compression for predicate evaluation has been hypothesized in the past but no solution has been presented [Pirk et al., 2014].

Early Pruning Extensions. In [Li et al., 2015], Li, Chasseur, and Patel look into lossless variable-length encoding schemes for Bit-Sliced Indices that are aimed at making the high-order bits informative in query processing. This solves the problem of data value skew and [Li et al., 2015] additionally exploits more frequently queried predicate values. If skew is heavy enough such that frequent values or frequently queried values would require less than 8 bits, than the resulting bit-sliced index would be faster for querying those values than a Column Sketch. Furthermore, as in traditional bit-sliced indices, the resulting index is helpful even when the column's code values are smaller than 8 bits.

However, [Li et al., 2015] does not consider lossy encoding schemes. By keeping the encoding schemes lossless, the padded variable length encoding schemes have larger memory footprint than a Column Sketch and have more expensive write times. As well, the resulting padded variable length encoding schemes are expensive to generate, with substantial run time spent on generating coding schemes for code sizes of 24 bits or less. An interesting line of future work is mixing the techniques from [Li et al., 2015] with Column Sketches, and using padded bitweaved columns inside the sketched column of a Column Sketch.

Lightweight Indexing. Lightweight data skipping techniques such as Zone Maps and their equivalent provide a way to skip over large blocks of data by keeping simple metadata such as min and max for each column. They are included in numerous recent systems [Francisco, 2011, Raman et al., 2013, Thusoo et al., 2010, Lamb et al., 2012, Xin et al., 2013] and how to best organize both the data and metadata is an area of ongoing research [Sun et al., 2014, Qin and Idreos, 2016, Metzger et al., 2005, Slezak et al., 2008, Sidirourgos and Kersten, 2013]. Amongst recent approaches, Column Imprints stands out as similar in nature to Column Sketches[Sidirourgos and Kersten, 2013]. Column Imprints also use histograms to better evaluate predicates, but do so for groups of values at a time instead of single values at at time.

For datasets with clustering properties, data skipping techniques notice that entire groups of values would evaluate the predicate to be either true or false, and so provide incredible speedup in these scenarios. For datasets that are not clustered, lightweight indices can't evaluate groups of data at a time and so scans need to check each value individually. In contrast, Column Sketches is able to handle these queries as it already works on an element by element basis. Thus, lightweight indices should be used in tandem with Column Sketches, as both have low update costs and target different scenarios.

Other Operators over Early Pruning Techniques. The use of early pruning techniques has been generalized to other operators beyond predicate evaluation [Feng and Lo, 2015, Pirk et al., 2014]. Most of this work can be applied to Column Sketches. For instance, a MAX operator can look at b high order bits and prune all records which are less than the maximum value seen using only those b bits, as they are certainly not the max. This is similar to Column Sketches, where any value that is not the maximum on the Column Sketch bytes is clearly not the maximum value in the column.

SIMD Scan Optimizations. There is a recent flurry of research on how to best integrate SIMD into column scans. We touch only on the main techniques and leave the reader to familiarize themselves with other work [Zhou and Ross, 2002, Feng et al., 2015, Lemire and Boytsov, 2015, Polychroniou et al., 2015]. The unpacking methods in this paper are based off work by Willhalm et al. where they use SIMD lanes to unpack codes one code per SIMD lane [Willhalm et al., 2013, 2009]. Improvements in the computational performance of scans is complementary to Column Sketches. Column Sketches are designed to be a scannable dense array structure, and so improvements in evaluating predicates apply equally to Column Sketches.

5.9 Conclusion

In this paper, we show that neither traditional indexing nor light-weight data skipping techniques provide performance benefits for queries with moderate selectivity over unclustered data. To provide performance improvements for this large class of queries, we introduce a new indexing technique, Column Sketches, which provides better scan performance regardless of data ordering, data distribution and query selectivity. Compared to state-of-the-art approaches for scan accelerators, Column Sketches are significantly easier to update and are more performant on scans over a range of differing data distributions. Possible extensions of Column Sketches include usage for operators other than equality and range predicates, such as aggregations, set inclusion predicates, and approximate query processing.

Chapter 6

Designing New Cerebral Data Structures

We've now seen multiple instances of Cerebral Data Structures and their ability to produce improvements in the space of data structures. Given these examples, we now return to two broader questions. First, how can the framework of Cerebral Data Structures be used to create new classes of data structure designs? And second, how does this work of Cerebral Data Structures integrate with approaches to automate the fitting of data structure to system?

6.1 Generating New Classes of Cerebral Data Structures

As mentioned in the introduction, generating new Cerebral Data Structures still requires human ingenuity, but the goal of the framework is that through adding structure to the creative process, we allow for greater clarity in thinking and the faster discovery of new designs. While prior sections provided end to end examples of new designs, this section is meant to highlight the design process and give ideas on how to generate new Cerebral Data Structure designs.

As a way to provide guidance, we break down several possible approaches to improving Cerebral Data Structures for hash tables. The goal is that this more in depth example also helps to consider how Cerebral Data Structures can be applied in new domains, such as say nearest neighbor indexing. The approaches introduced range from approaches which are immediately actionable, but will likely not have dramatic performance improvements, to open questions that would be full research projects to themselves but which would provide significant benefits for hash tables. Recall that the main metrics for hash tables are speed, memory consumption, and robustness and the components which contribute to the overall performance of hash tables are the hash function including its computational speed and collision probabilities, the storage of keys and values (which influences memory and also speed via cache misses), the method for handling collisions, and the cost of comparing keys which do collide. In Cerebral Data Structures, we generally focus on the components of greatest importance for data structures (step 1), but here we analyze each component to give a better idea on how to generate ideas for Cerebral Data Structures.

Comparison Operator. Analyzing how context can be integrated into the design and implementation of the comparison operator leads to fairly actionable ideas. The traditional method for comparing keys is to compare them in byte order for ordered keys such as strings or in an arbitrary order for keys which are tuples (step 2a). If data is skewed, as is often the case, it might be faster to compare keys in a different byte ordering which maximizes the probability of early termination. The goal would then be to introduce a formal parameterization of the context that allows for investigating how to make comparisons faster, and then to use this parameterization to design a new faster comparison operator which views bytes in a strategically chosen order (step 2b). The design of this comparison operator would then need to progress through the same stages as that of other Cerebral Data Structures: introducing equations for how the parametric model affects the performance of the comparison operator (step 2b), introducing estimators for the introduced parameters (step 3), and creating a decision procedure which takes in estimates of the parameters (and optionally runtime information) to produce a specific comparison function (step 4).

Collision Resolution. A second example to analyze is how context can be used to design and implement methods for collision resolution using the framework of Cerebral Data Structures. Currently, there are a range of methods for dealing with collision resolution, but they generally do not depend on the workload. For instance, separate chaining and linear probing tables generally put items at the first empty slot found (step 2a). In practice, it might be better to deal with collision resolution in a way that either makes items placement more predictable (so that these locations might be checked first) or skews more frequently queried items closer to the head of the checked locations. An example approach using this idea can be found in Knuth's Art of Computer Programming Vol. 3 [Knuth, 1998], wherein items with higher probability of being queried are put closer to the head of the linked list for separate chaining hash tables (step 2b). He analyzes how this affects the number of comparisons needed (step 2b), and what the optimal table is for this case (decision: more frequently queried items towards head of list, step 4). While not provided in the book, given a past sample of queries, simple counting histograms could be used to estimate which items are most likely to be queried (step 3). Alternative ideas could be used to handle this approach for linear probing tables, with the general ideas and steps likely being similar.

Hash Functions: Operations. While the previous two projects are more immediately actionable, we now cover several more complicated, but perhaps more impactful, ideas for generating more complex cerebral data structures for hashing. One idea is to use context in the design and implementation of the hash function to further analyze how to reduce hash computation compared to traditional approaches. For instance, while we already discussed in Chapter 3 how assumptions in traditional hashing mean they work over every byte of data, these assumptions also affect the computations done by hash functions on each byte (step 2a). An interesting direction of research is whether a parameterization of the workload can also be used to reduce the computation done on each byte. In many ways, the work of Mitzenmacher and Vadhan [Mitzenmacher and Vadhan, 2008] can be seen as achieving this already, as they show that 2-independent hash functions suffice when data is random enough compared to alternative approaches of using k-independent hash functions with $k > 2$ (steps 2b,4). Estimators of the Rényi entropy given in Chapter 3 could be used to estimate the needed entropy (step 3). An open question is whether alternative parameterizations lead to different computational benefits on the hash function while preserving enough uniformity for hash tables. For instance, current computational expenses for hash functions generally come as a result of "avalanche" steps at the end of computation (this is especially true for short keys), one interesting question is under what conditions on data can this step be removed. Such a parameterization would need to take in a likely modified idea of input randomness (step 2b), translate this into approximate uniformity of hash output, estimate this new form of randomness (step 3), and build the resulting hash function (step 4). The result would be faster hash tables for small data types such as integers and short strings.

Hash Functions: Collisions. For hash tables, one known way to improve performance is to decrease the collision probability coming from the hash function. The traditional approach is to assume hash outputs are distributed as i.i.d. uniform random variables, and then simply to deal with the collisions that occur (step 2a). While one might try to improve upon the output distribution, if each hash output is independent, then a uniform distribution produces the fewest collisions, and so no learned hash function can do better. However, one way to reduce the collision probability is by breaking this independence, something that has already been done for static data sets via approaches such as perfect hash functions [Fredman et al., 1984, Botelho et al., 2007, Dietzfelbinger et al., 1994, Belazzougui et al., 2009], and learned hashing techniques [Kraska et al., 2018, Sabek et al., 2021]. A (hard) open question is whether there are reasonable workload assumptions that can be made which allow for lower collision rates by breaking the independence of hash outputs while supporting efficient insertions (step 2b). The goal would be to find some parameterization of the workload that makes certain joint appearance probabilities likely, and to find hash functions which separate items from these parameterized workloads in a way that makes collisions less likely than i.i.d. uniform random variables. The additional steps would then need to be the same: create statistical estimators for this parameterization of the workload (step 3), and use these estimates plus runtime information to build hash functions (step 4).

Takeaways. This Section shows that even in the data structures analyzed in this thesis, there are many unexplored and realistic parameterizations of workloads that can be used to improve data structure performance. This is because by and large, data structures are still designed for arbitrary workloads and data distributions. Thus, by thinking about how parameters of workloads affect their functionality, and by capitalizing the increasing abundance of data about systems and their data structures, we can create and use better data structures.

6.2 Towards Automating Data Structure Design

The eventual goal of instance-optimized systems, and of data structures as a part of that vision, is to automate the creation of specifically tailored data structures that perfectly match the context they are deployed in. Already, we have seen aspects of this vision inside the framework for Cerebral Data Structures, wherein each of Entropy-Learned Hashing, Stacked Filters, and Column Sketches uses statistical estimators to gather information about context before optimization routines create data structures which are the best possible versions of Entropy-Learned Hash structures, Stacked Filters, or Column Sketches for the context at hand.

A much larger goal is to look beyond limited classes of data structures and to try find the best possible data structure for a context. While this is clearly a large vision, a road map towards that vision was created in The Data Calculator [Idreos et al., 2018], a project I was part of during my PhD. The main ideas behind the Data Calculator are in many ways similar to those of Cerebral Data Structures. Like Cerebral Data Structures, the Data Calculator tracks properties of its workload to try to create a parameterization of the workload. It then has a much larger search space of data structures, and an optimization algorithm which picks designs from this larger space of data structures.

6.2.1 An Overview of the Data Calculator

This Section gives a brief overview of the Data Calculator and its approach to automatically generating new data structure designs as combinations of existing designs. A more detailed description of the approach can be found in Idreos et al. [2018]. Here we focus on the relationship between the Data Calculator and its goal of overall automatic design of data structures with the framework of Cerebral Data Structures.

The Data Calculator, which currently tackles the problem of read-only key-value data structures, consists of three modules.

The first module is a parameterization of the workload, with this parameterization generally consisting of simpler workload parameters than those considered by Cerebral Data Structures. For instance, this might be parameters like the ratio of point gets (find key x) to range gets (find keys with $x_1 < \text{key} < x_2$) and histograms of the query values given.

The second module, and the most complex, is a space of *data layout primitives* and *data access primitives* that govern how a data structure lays out and accesses its data. The main idea behind the project is that combinations of data layout primitives and data access primitives define a data structure, and that by identifying these primitives we allow for novel data structures through combining them in new ways. As an example, B-trees are generally defined by their primary way of partitioning data (by approximate quantiles) but also have many more parameters such as their utilization ($> 50\%$) and how internal nodes are laid out (generally nodes are independently laid out in memory). Each of these choices is a primitive. To get to a new data structure, Cache Sensitive B-Trees [Rao and Ross, 2000], the internal nodes are no longer apart in memory but instead conjoined in a specific order for better cache locality when traversing the tree. This primitive, which is known as "sub-block physical layout", changes from "scattered" for B-trees to "breadth-first" to signify this change. The core concept behind the Data Calculator is that this idea of going between pointers containing different blocks to laying these blocks out in a specific order is not unique to B-trees, and that we can apply it more broadly and in a systematic manner if we have a language of data structures. Building off this is then that choices of data layout primitives then also provide a range of choices for data access primitives. The key idea here is that searching something like a sorted array has multiple options, one can do a linear scan, a binary search, an interpolation search, or another idea entirely. By grouping these together and allowing for algorithmic choice, we then also allow for different implementations of searches over sorted arrays to be easily conjoined with our ideas above on how to best store data. The list of data layout primitives can be found in Appendix D.

The third module is then a costing module which uses a library of learned models to take in a combination of data layout primitives, data access primitives, and the parameterization of the workload to produce an estimated cost of running this workload over the data structure. The list of learned models used can be found in Appendix D. Using these models then allows for questions like what is the cost of running this workload on a specific data structure idea I have in mind, as well as for more generic questions like what is the best data structure for this workload.

6.2.2 How Cerebral Data Structures adds to the vision of Automatic Data Structure Design

A major question is how these two frameworks for design and creation of data structures complement each other. In particular, the Data Calculator and Cerebral Data Structures contain many similarities: a parameterization of the workload, a search space of data structures, and an optimization routine to pick an optimal point in the search space (we note this architecture is not unique; for instance, query optimizers have the same components. However, its application to data structures brings new challenges).

When viewing the Data Calculator, and indeed any approach to automatic data structure creation, significant effort must be spent on how different approaches interact with each other so that the approaches can be combined. Thus, the central problems around an approach like the Data Calculator is creating the algebra of data structures, query routines over arbitrary points in this space, and calculating costs for points that are complex combinations of more primitive ideas. In this way, the effort in the Data Calculator goes into how existing ideas can be combined to create new combinations of those ideas (and thus entirely new data structures). At the same time, the design space of possible data structures formed by combinations of primitives is always incomplete. New ideas are continually created and enrich the space of possible data structure designs. The power of the Data Calculator is that once these new ideas are integrated, the new idea can be considered in conjunction with many other existing ideas immediately.

At the same time, the pace of inventing new design elements is extremely slow as most data structure designs published are (novel) combinations of existing design elements. This is where the contributions of the Cerebral Data structures framework come in the picture. Cerebral Data Structures can be thought of as a framework for expanding this design space by finding new ideas for data structure designs by more richly thinking about properties of workloads. For instance, when comparing the ways workloads are parameterized, Cerebral Data Structures have new parameterizations of workloads that are not represented in the Data Calculator (such as the entropy of individual bytes for hashing or more complex reasoning about the relationship between domain size and query skew for filter structures). These new parameterizations lead to new designs which are state-of-the-art for their respective data structures and whose designs are not covered by the Data Calculator. This presents a chance for feedback, wherein the new workload parameters identified by Cerebral Data Structures and new data structures created by Cerebral Data Structures are (eventually, and with significant effort) added back into the richer space of data structures encapsulated by the Data Calculator.

;

Chapter 7

Conclusion

In this dissertation, we proposed Cerebral Data Structures, a specific framework for advancing data structure design through proper use of context to design data structures. To back up our proposal, we showed several example data structures where creating formal parametric representations of context lead to better data structure designs. In particular, we showed for the produced data structures that this approach lead to both better performance and understandable performance, with the approaches having concrete performance equations that depended on the (interpretable) parameters of the workload. Additionally, we showed using this approach an expanded set of ways in which we could use learning in data structures, showing that context-awareness does not need to imply models inside a data structures. More specifically, we introduced three new Cerebral Data Structures: Entropy-Learned Hashing, Stacked Filters, and Column Sketches.

Entropy-Learned Hashing showed that we can parameterize the inherent randomness of incoming data in terms of its Rényi entropy, and then use this parameterization to make it so that we can hash only parts of incoming keys. As a result, hashing went from an operation whose computational costs depended on the size of the input key to one whose runtime is constant with respect to key size (assuming input keys have enough entropy). This produced up to 4$\times$ improvements in hashing-based tasks on medium-sized keys such as URLs over state-of-the-art solutions from Google and Meta, and up to 85$\times$ improvements on very large keys.

Stacked Filters introduced a simpler workload parameterization when compared to Learned filters, and which enabled capitalizing on workload knowledge in filter structures without the use of a classifier as part of the structure. When compared to traditional filters, Stacked Filters were able to get a 100× benefit in terms of the false positive rate at a given space constraint while keeping computational expense similar. When compared to learned filters, this approach of removing the classifier produced the same benefits in false positive rate vs. space tradeoffs but were up to 184× cheaper to query and provides robust behavior under workload shifts.

Column Sketches showed how estimation of the cumulative distribution function for data values combined with the creation of lossy auxiliary columns leads to efficient and robust scan performance. In particular, this approach created useful data representations such that each bit was useful for predicate evaluation, and then combined this new representation with a physical reorganization that guaranteed good scan performance. When compared to prior approaches, this had the benefit of both being up to 3× faster than the nearest competitor but also of producing good performance regardless of data values, data ordering, or the predicate in question.

Together, this set of results provides proof that introducing a parametric representation of workloads leads to fundamentally better performance on data structures of interest. The goal going forward is then to expand this approach to the many more classes of data structures that exist, in service of being able to create efficient and context-specialized data structures for data systems.

Bibliography

Genomic data science fact sheet. https://www.genome.gov/about-genomics/fact-sheets/Genomic-Data-Science.

Top 10 million websites: Openpagerank. "https://www.domcop.com/openpagerank/what-is-openpagerank".

Shalla secure services kg. http://www.shallalist.de.

Hacker news posts. https://www.kaggle.com/hacker-news/hacker-news-posts, 2015. Accessed: 2021-05-23.

Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 671–682, 2006. doi: 10.1145/1142473.1142548. URL http://doi.acm.org/10.1145/1142473.1142548.

Daniel J. Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013. doi: 10.1561/1900000024. URL http://dx.doi.org/10.1561/1900000024.

Jayadev Acharya, Alon Orlitsky, Ananda Theertha Suresh, and Himanshu Tyagi. Estimating renyi entropy of discrete distributions. *IEEE Transactions on Information Theory*, 63(1):38–56, 2017. doi: 10.1109/TIT.2016.2620435.

Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2O: A Hands-free Adaptive Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1103–1114, 2014. ISBN 9781450323765. doi: 10.1145/2588555.2610502. URL http://dl.acm.org/citation.cfm?id=2588555.2610502.

Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable bloom filters. *Inf. Process. Lett.*, 101(6):255–261, March 2007. ISSN 0020-0190.

Karl Anderson and Steve Plimpton. Firehose streaming benchmarks, 2015. "https://firehose.sandia.gov/".

Austin Appleby. murmurhash3. https://github.com/aappleby/smhasher/wiki/MurmurHash3, a. Accessed: 2021-05-23.

Austin Appleby. smhasher suite. https://github.com/aappleby/smhasher, b. Accessed: 2021-05-23.

Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2016. doi: 10.1145/2882903.2915231. URL https://www.cs.cmu.edu/{~}jarulraj/papers/2016.tile.sigmod.pdfhttp://dx.doi.org/10.1145/2882903.2915231.

Jean-Philippe Aumasson and Daniel J Bernstein. Siphash: a fast short-input prf. In *International Conference on Cryptology in India*, pages 489–508. Springer, 2012.

Eric Balkanski, Sharon Qian, and Yaron Singer. Instance specific approximations for submodular maximization. In *International Conference on Machine Learning*, pages 609–618. PMLR, 2021.

Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow.*, 7(1):85–96, September 2013. ISSN 2150-8097. doi: 10.14778/2732219.2732227. URL https://doi.org/10.14778/2732219.2732227.

Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, Tianchao Tim Li, Guy M. Lohman, Konstantinos Morfonios, René Müller, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Richard Sidle, Knut Stolze, and Sandor Szabo. Business Analytics in (a) Blink. *IEEE Data Engineering Bulletin*, 35(1):9–14, 2012. URL http://sites.computer.org/debull/A12mar/blink.pdf.

Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, 1(4):355–398, oct 1992. ISSN 1049-331X. doi: 10.1145/136586.136587. URL https://doi.org/10.1145/136586.136587.

Djamal Belazzougui, Fabiano C Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *European Symposium on Algorithms*, pages 682–693. Springer, 2009.

Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't Thrash: How to Cache Your Hash on Flash. *PVLDB*, (11):1627–1637, 2012.

Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. Bloom filters, adaptivity, and the dictionary problem. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 182–193, 2018. doi: 10.1109/FOCS.2018.00026.

Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based Order-preserving String Compression for Main Memory Column Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 283–296, 2009. doi: 10.1145/1559845.1559877. URL http://doi.acm.org/10.1145/1559845.1559877.

John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. Umac: Fast and secure message authentication. In *Annual International Cryptology Conference*, pages 216–233. Springer, 1999.

Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

Colin R. Blyth. Expected absolute error of the usual estimator of the binomial parameter. *The American Statistician*, 34(3):155–157, 1980. ISSN 00031305. URL http://www.jstor.org/stable/2683873.

Peter Boncz, Thomas Neumann, and Viktor Leis. Fsst: Fast random access string compression. 13 (12):2649–2661, 2020.

Fabiano C Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In *Workshop on Algorithms and Data Structures*, pages 139–150. Springer, 2007.

Alex D. Breslow and Nuwan S. Jayasena. Morton filters: Faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proc. VLDB Endow.*, 11(9):1041–1055, May 2018.

Andrei Broder, Michael Mitzenmacher, and Andrei Broder I Michael Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.

Andrei Z. Broder. On the resemblance and containment of documents. In Bruno Carpentieri, Alfredo De Santis, Ugo Vaccaro, and James A. Storer, editors, *SEQUENCES*, pages 21–29. IEEE, 1997. URL http://dblp.uni-trier.de/db/conf/sequences/sequences1997.html#Broder97.

Nathan Bronson and Xiao Shi. Open-sourcing f14 for faster, more memory-efficient hash tables. https://engineering.fb.com/2019/04/25/developer-tools/f14/.

Jehoshua Bruck, Jie Gao, and Anxiao Jiang. Weighted bloom filter. In *2006 IEEE International Symposium on Information Theory*, pages 2304–2308, 2006. doi: 10.1109/ISIT.2006.261978.

J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, page 106–112, New York, NY, USA, 1977. Association for Computing Machinery.

Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 59–65. ACM, 1978.

Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-Yeol Ahn, and Sue Moon. Analyzing the video popularity characteristics of large-scale user generated content systems. *IEEE/ACM Transactions on networking*, 17(5):1357–1370, 2009.

Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. Cosine: a cloud-cost optimized self-designing key-value storage engine. *Proceedings of the VLDB Endowment*, 15(1): 112–126, 2021.

Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 30–39. Society for Industrial and Applied Mathematics, 2004.

Jakub Chłędowski, Adam Polak, Bartosz Szabucki, and Konrad Tomasz Żołna. Robust learning-augmented caching: An experimental study. In *International Conference on Machine Learning*, pages 1920–1930. PMLR, 2021.

Charles Choi. Migrating big astronomy data to the cloud. *Nature*, 584:159–160, 08 2020. doi: 10.1038/d41586-020-02284-7.

Kai-Min Chung, Michael Mitzenmacher, and Salil Vadhan. Why simple hash functions work: Exploiting the entropy in a data stream. *Theory of Computing*, 9(30):897–945, 2013. doi: 10.4086/toc.2013.v009a030. URL http://www.theoryofcomputing.org/articles/v009a030.

Jeffrey S Cohen and Daniel M Kane. Bounds on the independence required for cuckoo hashing. *ACM Transactions on Algorithms*, 2009.

Yann Collet. xxhash. https://cyan4973.github.io/xxHash/. Accessed: 2021-05-23.

Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979. doi: 10.1145/356770.356776. URL http://doi.acm.org/10.1145/356770.356776.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009. ISBN 0262033844.

Andrew Crotty. Hist-tree: Those who ignore it are doomed to learn. In *CIDR*, 2021.

Zhenwei Dai and Anshumali Shrivastava. Adaptive learned bloom filter (ada-bf): Efficient utilization of the classifier with application to real-time information filtering on the web. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 11700–11710. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/86b94dae7c6517ec1ac767fd2c136580-Paper.pdf.

Niv Dayan and Moshe Twitto. Chucky: A succinct cuckoo filter for lsm-tree. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, page 365–378, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3457273. URL https://doi.org/10.1145/3448016.3457273.

Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 79–94. ACM, 2017.

Kyle Deeds, Brian Hentschel, and Stratos Idreos. Stacked filters: Learning to filter by structure. *Proceedings of the VLDB Endowment*, 14(4):600–612, dec 2020. ISSN 2150-8097.

Fan Deng and Davood Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 25–36. ACM, 2006.

Sarang Dharmapurikar, Praveen Krishnamurthy, and David E Taylor. Longest prefix matching using bloom filters. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 201–212. ACM, 2003.

Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer Auf Der Heide, Hans Rohnert, and Robert E Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997. ISSN 0196-6774.

Peter C Dillinger and Stefan Walzer. Ribbon filter: practically smaller than bloom and xor. *arXiv preprint arXiv:2103.02515*, 2021.

Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R Narasayya. Ai meets ai: Leveraging query executions to improve index recommendations. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1241–1258, 2019.

Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *arXiv preprint arXiv:2006.13282*, 2020.

Ogi Djuraskovic. Google search statistics and facts 2022, 2019. URL https://firstsiteguide.com/google-search-stats/.

Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. Quantifying tpc-h choke points and their optimizations. *Proc. VLDB Endow.*, 13(8):1206–1220, April 2020. ISSN 2150-8097. doi: 10.14778/3389133.3389138. URL https://doi.org/10.14778/3389133.3389138.

Aryeh Dvoretzky, Jack Kiefer, and Jacob Wolfowitz. Asymptotic minimax character of the sample distribution function and of the classical multinomial estimator. *The Annals of Mathematical Statistics*, pages 642–669, 1956.

Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88. ACM, 2014. https://github.com/efficient/cuckoofilter.

Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

Wenbin Fang, Bingsheng He, and Qiong Luo. Database compression on graphics processors. *Proceedings of the VLDB Endowment*, 3(1-2):670–680, 2010. doi: 10.14778/1920841.1920927. URL http://dx.doi.org/10.14778/1920841.1920927.

Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Engineering Bulletin*, 35(1):28–33, 2012.

Ziqiang Feng and Eric Lo. Accelerating aggregation using intra-cycle parallelism. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 291–302, 2015. doi: 10.1109/ICDE.2015.7113292. URL http://dx.doi.org/10.1109/ICDE.2015.7113292.

Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 31–46, 2015. doi: 10.1145/2723372.2747642. URL http://doi.acm.org/10.1145/2723372.2747642.

Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13(8):1162–1175, 2020.

P. Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics & Theoretical Computer Science*, pages 137–156, 2007.

Phil Francisco. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. *IBM Redbooks*, 2011. URL http://www.redbooks.ibm.com/redpapers/pdfs/redp4725.pdf.

Michael L Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984.

Dimitris Geneiatakis, Nikos Vrakas, and Costas Lambrinoudakis. Utilizing bloom filters for detecting flooding attacks against sip based services. *computers & security*, 28(7):578–591, 2009.

Arthur Gervais, Srdjan Capkun, Ghassan O Karame, and Damian Gruber. On the privacy provisions of bloom filters in lightweight bitcoin clients. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 326–335. ACM, 2014.

GNU Compiler Collection. gcc libstdc++ hash. https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/libsupc%2B%2B/hash_bytes.cc. Accessed: 2021-05-23.

Bob Goodwin, Michael Hopcroft, Dan Luu, Alex Clemmer, Mihaela Curmei, Sameh Elnikety, and Yuxiong He. Bitfunnel: Revisiting signatures for search. In *SIGIR '17 Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2017.

Google. Abseil Common Libraries. https://github.com/abseil/abseil-cpp.

Goetz Graefe. Modern B-Tree Techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011. URL http://dx.doi.org/10.1561/1900000028.

Goetz Graefe and Leonard D Shapiro. Data Compression and Database Performance. In *Proceedings of the ACM/IEEE-CS Symposium On Applied Computing (SAC)*, pages 22–27, 1991.

Thomas Mueller Graf and Daniel Lemire. Xor filters: Faster and smaller than bloom and cuckoo filters, 2019.

Jason Gregory. *Game engine architecture*. Taylor & Francis Ltd., 1 edition, 2009.

Shai Halevi and Hugo Krawczyk. Mmh: Software message authentication in the gbit/second rates. In *International Workshop on Fast Software Encryption*, pages 172–189. Springer, 1997.

Brian Hentschel, Michael S. Kester, and Stratos Idreos. Column sketches: A scan accelerator for rapid and robust predicate evaluation. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 857–872, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450347037. doi: 10.1145/3183713.3196911. URL https://doi.org/10.1145/3183713.3196911.

Brian Hentschel, Utku Sirin, and Stratos Idreos. Entropy-learned hashing: Constant time hashing with controllable uniformity. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, New York, NY, USA, 2022. Association for Computing Machinery.

Stratos Idreos. *Database Cracking: Towards Auto-tuning Database Kernels.* PhD thesis, University of Amsterdam, 2010.

Stratos Idreos and Mark Callaghan. Key-value storage engines. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2667–2672, 2020.

Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *In CIDR*, 2007.

Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S Kester, and Demi Guo. The data calculator: Data structure design and cost synthesis from first principles and learned cost models. In *Proceedings of the 2018 International Conference on Management of Data*, pages 535–550, 2018.

Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. Design continuums and the path toward self-designing key-value stores that know and learn. In *CIDR*, 2019.

Intel. Intel VTune Amplifier XE Performance Profiler, 2021. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. Row-wise Parallel Predicate Evaluation. *Proceedings of the VLDB Endowment*, 1(1):622–634, 2008. URL http://www.vldb.org/pvldb/1/1453925.pdf.

Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe? In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 715–730, 2017. ISBN 9781450341974. doi: 10.1145/3035918.3064049. URL http://dl.acm.org/citation.cfm?doid=3035918.3064049.

Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Radixspline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–5, 2020.

Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: building a better bloom filter. In *European Symposium on Algorithms*, pages 456–467. Springer, 2006.

Don Knuth. Notes on "open" addressing, 1963.

Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching.* Addison Wesley Longman Publishing Co., Inc., USA, 1998. ISBN 0201896850.

Chun-Wa Ko, Jon Lee, and Maurice Queyranne. An exact algorithm for maximum entropy sampling. *Operations Research*, 43(4):684–691, 1995.

Onur Kocberber, Babak Falsafi, and Boris Grot. Asynchronous memory access chaining. *Proc. VLDB Endow.*, 9(4):252–263, 2015.

Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

Tim Kraska, Mohammad Alizadeh, and Alex Beutel. H. chi, jialin ding, ani kristo, guillaume leclerc, samuel madden, hongzi mao, and vikram nathan. 2019. sagedb: A learned database system. In *9th Conference on Innovative Data Systems Research, CIDR*, 2019.

Matt Kulukundis. Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step. https://www.youtube.com/watch?v=ncHmEUmJZf4.

Andrew Lamb, Matt Fuller, and Ramakrishna Varadarajan. The Vertica Analytic Database: C-Store 7 Years Later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012. URL http://dl.acm.org/citation.cfm?id=2367518.

Harald Lang, Thomas Neumann, Alfons Kemper, and Peter Boncz. Performance-optimal filtering: Bloom overtakes cuckoo at high throughput. *Proceedings of the VLDB Endowment*, 12(5):502–515, 2019.

David J. Lee, Samuel McCauley, Shikha Singh, and Max Stein. Telescoping filter: A practical adaptive filter. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 60:1–60:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Journal of Software: Practice and Experience*, 45(1):1–29, 2015. doi: 10.1002/spe.2203. URL http://dx.doi.org/10.1002/spe.2203.

Yinan Li and Jignesh M Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 289–300, 2013. doi: 10.1145/2463676.2465322. URL http://doi.acm.org/10.1145/2463676.2465322.

Yinan Li, Craig Chasseur, and Jignesh M Patel. A Padded Encoding Scheme to Accelerate Scans by Leveraging Skew. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1509–1524, 2015. doi: 10.1145/2723372.2737787. URL http://doi.acm.org/10.1145/2723372.2737787.

Linux. Perf Wiki, 2021. https://perf.wiki.kernel.org/.

Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. In *International Conference on Machine Learning*, pages 6237–6247. PMLR, 2020.

Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 334–350, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359566. doi: 10.1145/3341302.3342076. URL https://doi.org/10.1145/3341302.3342076.

Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. *Bao: Making Learned Query Optimization Practical*, page 1275–1288. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450383431. URL https://doi.org/10.1145/3448016.3452838.

Pascal Massart. The tight constant in the dvoretzky-kiefer-wolfowitz inequality. *The annals of Probability*, pages 1269–1283, 1990.

John K. Metzger, Barry M. Zane, and Foster D. Hinshaw. Limiting scans of loosely ordered and/or grouped relations using nearly ordered maps, 2005. URL https://www.google.com/patents/US6973452.

Michael Mitzenmacher. A model for learned bloom filters and optimizing by sandwiching. In *Advances in Neural Information Processing Systems*, pages 464–473, 2018.

Michael Mitzenmacher and Salil Vadhan. Why simple hash functions work: Exploiting the entropy in a data stream. pages 746–755, 01 2008.

Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. Adaptive cuckoo filters. *ACM J. Exp. Algorithmics*, 25, March 2020. ISSN 1084-6654. doi: 10.1145/3339504. URL https://doi.org/10.1145/3339504.

Michael David Mitzenmacher and Alistair Sinclair. *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, 1996. AAI9723118.

Guido Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 476–487, 1998. URL http://dl.acm.org/citation.cfm?id=645924.671173.

Hamid Mohamadi, Justin Chu, Benjamin P. Vandervalk, and Inanc Birol. ntHash: recursive nucleotide hashing. *Bioinformatics*, 32(22):3492–3494, 07 2016. ISSN 1367-4803. doi: 10.1093/bioinformatics/btw397. URL https://doi.org/10.1093/bioinformatics/btw397.

Ingo Müller, Cornelius Ratsch, and Franz Faerber. Adaptive string dictionary compression in in-memory column-store database systems. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 283–294, 2014.

George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. An analysis of approximations for maximizing submodular set functions—i. *Mathematical programming*, 14(1):265–294, 1978.

Hyeonwoo Noh, Andre Araujo, Jack Sim, and Bohyung Han. Large-scale image retrieval with attentive deep local features. *International Conference on Computer Vision (ICCV)*, 2016. URL http://arxiv.org/abs/1612.06321.

Maciej Obremski and Maciej Skorski. Renyi entropy estimation revisited. In Klaus Jansen, José D. P. Rolim, David Williamson, and Santosh S. Vempala, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2017, August 16-18, 2017, Berkeley, CA, USA*, volume 81 of *LIPIcs*, pages 20:1–20:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

Patrick E. O'Neil and Dallan Quass. Improved query performance with variant indexes. *ACM SIGMOD Record*, 26(2):38–49, 1997. URL http://dl.acm.org/citation.cfm?id=253262.253268.

Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996. URL http://dblp.uni-trier.de/db/journals/acta/acta33.html#ONeilCGO96.

Anna Pagh, Rasmus Pagh, and Milan Ružić. Linear probing with 5-wise independence. *SIAM Rev.*, 53(3):547–558, August 2011. ISSN 0036-1445. doi: 10.1137/110827831. URL https://doi.org/10.1137/110827831.

Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, may 2004. ISSN 01966774. doi: 10.1016/j.jalgor.2003.12.002. URL http://dl.acm.org/citation.cfm?id=1006424.1006426.

Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 775–787. ACM, 2017. https://github.com/splatlab/cqf.

Mihai Pătraşcu and Mikkel Thorup. On the k-independence required by linear probing and minwise independence. In *Automata, Languages and Programming*, pages 715–726, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

Mihai Patrascu and Mikkel Thorup. The power of simple tabulation hashing. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, page 1–10, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306911. doi: 10.1145/1993636.1993638. URL https://doi.org/10.1145/1993636.1993638.

W. W. Peterson. Addressing for random-access storage. *IBM Journal of Research and Development*, 1(2):130–146, 1957. doi: 10.1147/rd.12.0130.

Geoff Pike and Jyrki Alakuijala. Cityhash, Jan 2011. https://github.com/google/cityhash.

Geoff Pike and Jyrki Alakuijala. Farmhash, Apr 2014. https://github.com/google/farmhash.

Holger Pirk, Stefan Manegold, and Martin Kersten. Waste not... Efficient co-processing of relational data. *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 508–519, 2014. doi: doi.ieeecomputersociety.org/10.1109/ICDE.2014.6816677.

Orestis Polychroniou and Kenneth A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 755–766, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450323765. doi: 10.1145/2588555.2610522. URL https://doi.org/10.1145/2588555.2610522.

Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1493–1508, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450327589. doi: 10.1145/2723372.2747645. URL https://doi.org/10.1145/2723372.2747645.

M. J. D. Powell. *A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation*, pages 51–67. 1994.

Wilson Qin and Stratos Idreos. Adaptive Data Skipping in Main-Memory Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2255–2256, 2016. doi: 10.1145/2882903.2914836. URL http://doi.acm.org/10.1145/2882903.2914836.

Do Le Quoc, Istemi Ekin Akkus, Pramod Bhatotia, Spyros Blanas, Ruichuan Chen, Christof Fetzer, and Thorsten Strufe. Approxjoin: Approximate distributed joins. In *ACM Symposium of Cloud Computing (SoCC) 2018*, 2018.

Jack W Rae, Sergey Bartunov, and Timothy P Lillicrap. Meta-learning neural bloom filters. *arXiv preprint arXiv:1906.04304*, 2019.

M. V. Ramakrishna. Hashing practice: Analysis of hashing and universal hashing. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, page 191–199, New York, NY, USA, 1988. Association for Computing Machinery. ISBN 0897912683. doi: 10.1145/50202.50223. URL https://doi.org/10.1145/50202.50223.

M. V. Ramakrishna. Practical performance of bloom filters and parallel free-text searching. *Commun. ACM*, 32(10):1237–1239, October 1989. ISSN 0001-0782. doi: 10.1145/67933.67941. URL https://doi.org/10.1145/67933.67941.

Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., USA, 3 edition, 2002. ISBN 0072465638.

Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. Constant-Time Query Processing. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 60–69, 2008. doi: 10.1109/ICDE.2008.4497414. URL http://dx.doi.org/10.1109/ICDE.2008.4497414.

Vijayshankar Raman, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, Liping Zhang, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, and Shaorong Liu. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *Proceedings of the VLDB Endowment*, 6(11):1080–1091, 2013. ISSN 21508097. doi: 10.14778/2536222.2536233. URL http://dl.acm.org/citation.cfm?id=2536222.2536233.

Sukriti Ramesh, Odysseas Papapetrou, and Wolf Siberski. Optimizing distributed joins with bloom filters. In *International Conference on Distributed Computing and Internet Technology*, pages 145–156. Springer, 2008.

Jun Rao and Kenneth A. Ross. Making B+-trees Cache Conscious in Main Memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 475–486, 2000. ISBN 1581132174. doi: 10.1145/342009.335449. URL http://dl.acm.org/citation.cfm?id=342009.335449.

David Reinsel, John Gantz, and John Rydning. The digitization of the world: From edge to core, 2018. URL https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.

Stefan Richter, Victor Alvarez, and Jens Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proc. VLDB Endow.*, 9(3):96–107, November 2015. ISSN 2150-8097. doi: 10.14778/2850583.2850585. URL https://doi.org/10.14778/2850583.2850585.

Meghan Rimol. Gartner forecasts worldwide it spending to grow 5.1% in 2022. https://www.gartner.com/en/newsroom/press-releases/2022-01-18-gartner-forecasts-worldwide-it-spending-to-grow-five-point-1-percent-in-2022, 2022.

RocksDB. Rocksdb trace, replay, analyzer, and workload generation, 2020. "https://github.com/facebook/rocksdb/wiki/RocksDB-Trace,-Replay,-Analyzer,-and-Workload-Generation".

Kenneth A. Ross. Efficient hash probes on modern processors. In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 1297–1301. IEEE Computer Society, 2007.

Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, and Tim Kraska. When are learned models better than hash functions? *arXiv preprint arXiv:2107.01464*, 2021.

Jeanette P Schmidt and Alan Siegel. The analysis of closed hashing under limited randomness. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 224–234, 1990.

P. Griffiths Selinger, Morton M. Astrahan, Donald D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979. ISBN 089791001X. doi: 10.1145/582095.582099. URL http://dl.acm.org/citation.cfm?id=582095.582099.

Lefteris Sidirourgos and Martin L. Kersten. Column Imprints: A Secondary Index Structure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 893–904, 2013. URL http://dl.acm.org/citation.cfm?id=2463676.2465306.

Utku Sirin and Anastasia Ailamaki. Micro-architectural analysis of olap: Limitations and opportunities. *Proc. VLDB Endow.*, 13(6):840–853, 2020.

Dominik Slezak, Jakub Wroblewski, Victoria Eastwood, and Piotr Synak. Brighthouse: an analytic data warehouse for ad-hoc queries. *Proceedings of the VLDB Endowment*, 1(2):1337–1345, 2008. URL http://www.vldb.org/pvldb/1/1454174.pdf.

Benjamin Spector, Andreas Kipf, Kapil Vaidya, Chi Wang, Umar Farooq Minhas, and Tim Kraska. Bounding the last mile: Efficient learned string indexing. *arXiv preprint arXiv:2111.14905*, 2021.

Zachary Stephens, Skylar Lee, Faraz Faghri, Roy Campbell, Chengxiang Zhai, Miles Efron, Ravishankar Iyer, Michael Schatz, Saurabh Sinha, and Gene Robinson. Big data: Astronomical or genomical? *PLoS biology*, 13:e1002195, 07 2015. doi: 10.1371/journal.pbio.1002195.

Oracle ZFS Steve Tunstall. Dedupe 2.0. https://blogs.oracle.com/wonders-of-zfs-storage/dedupe-20-v2, 2017. Accessed: 2021-05-23.

Henrik Stranneheim, Max Käller, Tobias Allander, Björn Andersson, Lars Arvestad, and Joakim Lundeberg. Classification of dna sequences using bloom filters. *Bioinformatics*, 26(13):1595–1600, 2010.

Danny Sullivan. Google now handles at least 2 trillion searches per year, 2016. URL https://searchengineland.com/google-now-handles-2-999-trillion-searches-per-year-250247.

Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. Fine-grained Partitioning for Aggressive Data Skipping. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1115–1126, 2014. doi: 10.1145/2588555.2610515. URL http://doi.acm.org/10.1145/2588555.2610515.

Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2012.

Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 996–1005, 2010. doi: 10.1109/ICDE.2010.5447738. URL http://dx.doi.org/10.1109/ICDE.2010.5447738.

Kapil Vaidya, Eric Knorr, Tim Kraska, and Michael Mitzenmacher. Partitioned learned bloom filter. *arXiv preprint arXiv:2006.03176*, 2020.

Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, pages 1009–1024, 2017.

Xiujun Wang, Yusheng Ji, Zhe Dang, Xiao Zheng, and Baohua Zhao. Improved weighted bloom filter and space lower bound analysis of algorithms for approximated membership querying. In Matthias Renz, Cyrus Shahabi, Xiaofang Zhou, and Muhammad Aamir Cheema, editors, *Database Systems for Advanced Applications*, pages 346–362, Cham, 2015.

Yi Wang, Diego Barrios Romero, Daniel Lemire, and Li Jin. Modern non-cryptographic hash function and pseudorandom generator. 2020.

Jan Wassenberg and Peter Sanders. Engineering a multi-core radix sort. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, page 160–169. Springer-Verlag, 2011. ISBN 9783642233968.

Mark N. Wegman and J.Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, 1981. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(81)90033-7. URL https://www.sciencedirect.com/science/article/pii/0022000081900337.

Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The implementation and performance of compressed databases. *SIGMOD Rec.*, 29(3):55–67, 2000. ISSN 0163-5808. doi: 10.1145/362084.362137. URL http://doi.acm.org/10.1145/362084.362137.

Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009. URL http://www.vldb.org/pvldb/2/vldb09-327.pdf.

Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. Vectorizing database column scans with complex predicates. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 1–12, 2013. URL http://www.adms-conf.org/2013/muller_adms13.pdf.

Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and Rich Analytics at Scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 13–24, 2013. doi: 10.1145/2463676.2465288. URL http://doi.acm.org/10.1145/2463676.2465288.

Oracle ZFS. Zfs deduplication. https://blogs.oracle.com/bonwick/zfs-deduplication-v2, 2019. Accessed: 2021-05-23.

Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1567–1581, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335317. doi: 10.1145/2882903.2915222. URL https://doi.org/10.1145/2882903.2915222.

Yanxia Zhang and Yongheng Zhao. Astronomy in the big data era. *Data Science Journal*, 14:1–9, 05 2015. doi: 10.5334/dsj-2015-011.

Tianqi Zheng, Zhibin Zhang, and Xueqi Cheng. Saha: A string adaptive hash table for analytical databases. *Applied Sciences*, 10(6), 2020. ISSN 2076-3417. doi: 10.3390/app10061915. URL https://www.mdpi.com/2076-3417/10/6/1915.

Jingren Zhou and Kenneth A. Ross. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 145–156, 2002. doi: 10.1145/564691.564709. URL http://doi.acm.org/10.1145/564691.564709.

YongKang Zhu. Linker throughput improvement in visual studio 2019, 2019. https://devblogs.microsoft.com/cppblog/linker-throughput-improvement-in-visual-studio-2019/.

Zichen Zhu, Ju Hyoung Mun, Aneesh Raman, and Manos Athanassoulis. Reducing bloom filter cpu overhead in lsm-trees on modern storage devices. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)*, DAMON'21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385565. doi: 10.1145/3465998.3466002. URL https://doi.org/10.1145/3465998.3466002.

A. Zobrist. A new hashing method with application for game playing. *ICGA Journal*, 13:69–73, 1990.

Marcin Zukowski, Peter A. Boncz, and Sándor Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin*, 28(2):17–22, 2005.

Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, page 59, 2006. doi: 10.1109/ICDE.2006.150. URL http://dx.doi.org/10.1109/ICDE.2006.150.

Appendix A

Appendix: Linear Probing Proofs

A.1 Overview and Results

As in the main document, we start our proof by going over full-key hashing, then analyze partial-key hashing with a fixed dataset, and finally cover partial-key hashing with random data. Before starting the proof, we briefly cover the results. When the multi-set $S_{|L}$ is fixed, the expectation is an expectation over the randomness of the hash function H. When it is not fixed, the expectation is taken over both the random multi-set $S_{|L}$ and the random hash function H.

For full-key hashing, given any set of n keys and a hash table with m slots, the expected number of comparisons for querying a key not in the dataset and the average number of comparisons when querying for an existing key are:

$$\mathbb{E}[P'] \leq \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2}) \tag{A.1}$$

$$\mathbb{E}[P] \leq \frac{1}{2}(1 + \frac{1}{1-\alpha}) \tag{A.2}$$

Letting $c = \sum_x z_x^2$ be the number of collisions in $S_{|L}$ and $d = \sum_{x:z_x \geq 2} z_x$ the number of duplicated keys in $S_{|L}$, the costs for partial-key hashing are

$$\mathbb{E}[P'] \leq \begin{cases} \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2} + \sum_{x \neq y} \frac{z_x^2}{m(1-\alpha)^2}) & \text{if } z_y = 0 \\ \frac{z_y}{1-\alpha} + \frac{1}{(1-\alpha)^2} + \sum_{x \neq y} \frac{z_x^2}{m(1-\alpha)^2} & \text{if } z_y > 0 \end{cases} \tag{A.3}$$

$$\begin{aligned} \mathbb{E}[P] &\leq \frac{n-d}{2n} + \frac{1}{2}Q_0(m,n) + \frac{c}{m}Q_0(m,n) + \frac{c+d}{2n}Q_0(m,d) \\ &\approx (\frac{1}{2} + \frac{c}{n})(1 + \frac{1}{1-\alpha}) \end{aligned} \tag{A.4}$$

where here y is the queried for key in P' and the approximation in (A.4) uses the assumption that d is small compared to m. For random data from an i.i.d. source with Renyi entropy H_2, these translate to the following bounds:

$$\mathbb{E}[P'] \leq \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2}) + n2^{-H_2(L(X))}\frac{3}{2(1-\alpha)^2}$$

$$\mathbb{E}[P] \leq \frac{1}{2}(1 + \frac{1}{1-\alpha}) + n2^{-H_2(L(X))}(1 + \frac{1}{1-\alpha})$$

Comparing Partial-Key Hashing with Full-Key Hashing. The above equations, when fully analyzed, make a rather intuitive point: when there are only a few duplicates in K_L so that it closely approximates full-key hashing, the number of comparisons required by full-key and partial-key hashing are close. The general form of each equation for partial key hashing is that it is the same as full-key hashing plus a small penalty term. These penalty terms go to 0 as the keys in partial-key hashing become more distinct.

For all equations, we note the equations above are relatively tight bounds as long as α is not close to 1. This is reasonable to assume as all hash tables keep $\alpha < 0.9$ in practice because of the extremely bad behavior of linear probing tables as $\alpha \to 1$, wherein they degrade to linear scans. The most common range for α is between 0.25 and 0.85, with higher alpha leading to better memory usage but slower query times.

A.2 Analysis of Full Key Linear Probing

To prove (A.3) and (A.4), we start by proving (A.1) and (A.2), first proven by Donald Knuth. Our proof initially follows steps taken in The Art of Computer Programming[Knuth, 1998], however we deviate from the proof given there because that approach does not generalize to partial-key hashing. Additionally, we believe this proof is simpler to follow than the proof in [Knuth, 1998].

Counting Hash Sequences. For n distinct items, we have m^n possible ways to assign them to $[m] = \{0, \dots, m-1\}$, all of which are equally likely by our hashing assumptions. Of all possible hash sequences, we first consider how many leave position 0 empty in the hash table.

Theorem A.2.1. *The number of hash sequences such that n items are hashed into $[1, m-1]$, and no overflow occurs so that position 0 is empty is*

$$\begin{cases} (1 - \frac{n}{m}) \cdot h(m, n) & \text{if } 0 \leq n \leq m \\ 0 & \text{otherwise} \end{cases} \tag{A.5}$$

where $h(m, n)$ is the number of ways to hash n objects into $[0, m-1]$.

This is a slight reformulation of the way it is stated in [Knuth, 1998] so that it generalizes to partial-key hashing. To prove the theorem, note that hash sequences which hash n items into $[1, m-1]$ and do not overflow into position 0 are precisely the same as those that leave position 0 empty when hashing into a hash table of size m and resolving collisions via linear probing. This probability is $1 - \frac{n}{m}$ by the circular symmetry of linear probing. Thus, for full-key hashing there are $(1 - \frac{n}{m})m^n$ sequences which leave location 0 empty.

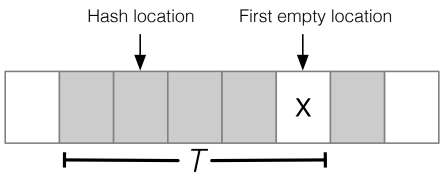
The next theorem gives the probability that a run of occupied slots begins at location 1 in the table and stops at location k.

Theorem A.2.2. *Let $g(m, n, k)$ be the number of sequences when hashing into a hash table of size m that leave position 0 empty, positions 1 to $k-1$ filled, and k empty. This is*

$$g(m, n, k) = \binom{n}{k-1} \frac{1}{k} \frac{m-n-1}{m-k} h(k, k-1) h(m-k, n-k+1) \tag{A.6}$$

This theorem simply builds off of Theorem A.2.1 as there are $\binom{n}{k-1}$ ways to choose the $k-1$ elements in the run starting at 1, and then by Theorem A.2.1 there are $\frac{1}{k}h(k,k-1)$ and $\frac{m-n-1}{m-k}h(m-k,n-k-1)$ ways to create the two subsequences with locations $0,k$ empty for each choice of the $k-1$ elements.

Analyzing Chain Length. We use the two theorems above to analyze a random variable T which represents the length counting from the empty position which a new item is inserted into to the final occupied position before the next empty position (going from left to right). The figure below shows an example.



If we know the expected value for T for a new item to be inserted, we can get its average distance from its hash location by the uniform randomness of hashing. Namely, for any chain of length T, each location in the chain is equally likely as an initial hash location, so $\mathbb{E}[P'|T] = \frac{1}{T}\sum_{i=1}^{T} i = \frac{1}{2}+\frac{1}{2}T$. It follows that

$$\mathbb{E}[P'] = \mathbb{E}[\mathbb{E}[P'|T]] = \frac{1}{2} + \frac{1}{2}\mathbb{E}[T]$$

To see how the two theorem above apply to our analysis of T, assume that the insertion key hashes to location a. If the insertion key is hashed into a chain of length t, then for some $t \geq 1, 0 \leq k \leq t-1$, we have $a-k$ empty, $a-k+1,\ldots,a+(t-k-1)$ full, and $a+(t-k)$ empty. We note by symmetry that the same number of sequences produce a chain of this nature as which produce a chain such that 0 is empty, $1,\ldots,t-1$ is full, t is empty. Since we have t choices for k, it then follows that there are $t\cdot g(m,n,t)$ hash sequences for which the insertion item is inserted into a chain of length t.

The approach taken by Knuth is to analyze $E[T] = \sum_{t\geq 1} t^2 g(m,n,t)$ directly by using Abel's Binomial Theorem, i.e.

$$(x+y)^n = \sum_k \binom{n}{k} x(x-kz)^{k-1}(y+kz)^{n-k}$$

While correct, this approach feels magical as the use of Abel's Binomial is unintuitive (at least to the authors). Additionally, it does not generalize to the case of partial-key hashing, and so we take a different approach.

Analyzing Chain Length By Connecting Chain Length and the Probability of Chain Inclusion. Our approach to analyze $\mathbb{E}[T]$ is to look at the probability that each item is in the same probe chain as the inserted item. Namely, if we let $x_1,\ldots,x_n$ be the keys inserted into the hash table, then we have that

$$\mathbb{E}[T] = 1 + \sum_{i=j}^{n} P(x_j \in T)$$

Here we slightly abuse notation to have T both represent the length of the chain on the left and the set of objects in the chain on the right. We now connect this probability that $x_j \in T$, which is equal for all j, to the expected chain length conditioned on a single new item x being part of the chain T.

More concretely, let T_i be the chain length of an inserted item if i of the n items in the hash table have the same hash value as the item to be inserted. All other items are distributed uniformly at random and independently of the hash value of the to be inserted item. The following theorem holds.

Theorem A.2.3. *If C represents a set of i values which are fixed to have the same hash location as the newly inserted item, then for $x \notin C$, we have*

$$P(x \in T_i) = \frac{1}{m}\,\mathbb{E}[T_{i+1}]$$

Proof. We first derive the formula for $\mathbb{E}[T_i]$. Assume as before that the insertion key hashes to location a. If the new item is hashed into a chain of length t, then for some $t \geq 1, 0 \leq k \leq t-1$, we have $a-k$ empty, $a-k+1, \ldots, a+(t-k-1)$ full, and $a+(t-k)$ empty. Again by symmetry, the number of sequences that produce a chain of this nature is the same as the number of sequences that produce a chain such that 0 is empty, $1, \ldots, t-1$ is full, t is empty, and the i items hash to location k. Let this number be $g(m,n,t,i,k)$. Then $f(m,n,t,i) = \sum_{k\geq 0} g(m,n,t,i,k)$ is the total number of sequences such that the new item is hashed into a chain of length t.

The value for $f(m,n,t,i)$ is calculable directly as this is identical to hashing $t-i-1$ items of multiplicity 1 and 1 value of multiplicity i into the first t slots keeping slot 0 empty, and hashing $n-(t-1)$ items into the final $m-t$ slots. Using theorem A.2.1, there are $\frac{1}{t}h(t,t-i)$ ways to do the first and $\frac{m-n-1}{m-t}h(m-t,n+1-t)$ ways to do the second, and $\binom{n-i}{t-i-1}$ ways to which $t-i-1$ elements from $K \setminus C$ are the elements in the run starting at 1. It follows that

$$f(m,n,t,i) = \binom{n-i}{t-i-1}\frac{1}{t}\frac{m-n-1}{m-t}h(t,t-i)h(m-t,n-t+1)$$

The expected value of T_i is then $m^{-(n-i)}\sum_{t\geq 1} t \cdot f(m,n,t,i)$.

We now calculate the probability that $x \notin C$ is in T_i. To do so, we count the number of chains which contain x, and then divide by the total number of possible hash sequences. Using the same logic as before, for a key hashing to location a, the number of chains containing x is equivalent to the number of chains such that 0 is empty, $1, \ldots, t-1$ is full and contains x, t is empty, and i items hash to location k. For a chain of length t, this means we have $\binom{n-i-1}{t-i-2}$ choices for the elements in the run between 0 and t, and we have $\frac{1}{t}h(t,t-i)$ and $\frac{m-n-1}{m-t}h(m-t,n-t+1)$ ways to hash these items to chain 1 and chain 2 respectively such that 0 and t are empty. Summing across all possible chain lengths, the number of chains which contain $x \notin C$ is

$$\begin{aligned} &= \sum_t \binom{n-i-1}{t-i-2}\frac{1}{t}\frac{m-n-1}{m-t}h(t,t-i)h(m-t,n-t+1) \\ &= \sum_t t f(m,n,t,i+1) \end{aligned}$$

where we use that $h(t,t-i-1) = t \cdot h(t,t-i)$. Dividing by the number of hash sequences, $m^{n-i} = m^{n-(i+1)}m$ gives that $P(x \in T_i) = \frac{1}{m}\,\mathbb{E}[T_{i+1}]$. □

Finishing the Proof. Using these relationship between $\mathbb{E}[T_i]$ and $P(x \in T_i)$, we have

$$\begin{aligned} E[T] &= 1 + \sum_{i=1}^{n} P(x_i \in T_0) \\ &= 1 + \frac{n}{m}\mathbb{E}[T_1] \\ &= 1 + \frac{n}{m}(2 + \frac{n-1}{m}\mathbb{E}[T_2])) \end{aligned}$$

where here we use that $\mathbb{E}[T_i] = (i+1) + \sum_{x \notin C} P(x \in T_i)$. If we continue expanding the values of $\mathbb{E}[T_i]$, we get that

$$\begin{aligned} \mathbb{E}[T] &= 1 + 2\frac{n}{m} + 3\frac{n^{\underline{2}}}{m^2} + \cdots + \frac{(n+1)n!}{m^n} \\ &= Q_1(m,n) \end{aligned}$$

where

$$Q_r(m,n) = \sum_{k \geq 0} \binom{k+r}{k} \frac{n^{\underline{k}}}{m^k}$$

It follows that

$$\mathbb{E}[P'] = \frac{1}{2} + \frac{1}{2} Q_1(m,n)$$

recovering the value proven in [Knuth, 1998]. Noting that $\frac{n^{\underline{k}}}{m^k} \leq \alpha^k$ and using that $\sum_{k \geq 0}(k+1)\alpha^k = d/d\alpha(\sum_{k \geq 0} \alpha^k) = (1-\alpha)^{-2}$ gives the desired bound $\mathbb{E}[P'] \leq \frac{1}{2}(1 + (1-\alpha)^{-2})$

Average Cost to Query an existing item. To go from $\mathbb{E}[P']$, the cost for a newly inserted key, to $\mathbb{E}[P]$, the average cost to query a key in the table, we average over the cost to insert the first n keys. Because the cost of an unsuccessful search in linear probing is the same as the cost to insert a key, this is the same as averaging over unsuccessful searches over tables storing 1 through n elements. So $\mathbb{E}[P] = \frac{1}{n}\sum_{i \geq 0}^{n} \mathbb{E}[P'_i]$ where $\mathbb{E}[P'_i]$ is the expected cost to insert a key if there are i items in the table. Thus

$$\begin{aligned} \mathbb{E}[P] &= \frac{1}{2} + \frac{1}{2n} \sum_{i=0}^{n-1} Q_1(m,i) \\ &= \frac{1}{2} + \frac{1}{2n} \sum_{i=0}^{n-1} \sum_{k \geq 0} (k+1) \frac{i^{\underline{k}}}{m^k} \\ &= \frac{1}{2} + \frac{1}{2n} \sum_{k \geq 0} \sum_{i=0}^{n-1} (k+1) \frac{i^{\underline{k}}}{m^k} \end{aligned}$$

Now we use the rules of "finite calculus", which says that if $\Delta f(x) = f(x+1) - f(x) = g(x)$, then $\sum_{i=a}^{b} g(x) = f(b+1) - f(a)$. Here we have $\Delta x^{\underline{k}} = kx^{\underline{k-1}}$, so $\sum_{x=0}^{n-1} kx^{\underline{k-1}} = n^{\underline{k}}$. Plugging this in above, we have

$$\begin{aligned}\mathbb{E}[P] &= \frac{1}{2} + \frac{1}{2n}\sum_{k\geq 0}\frac{n^{\underline{k+1}}}{m^k} \\ &= \frac{1}{2} + \frac{1}{2}Q_0(m, n-1)\end{aligned}$$

Using the same bound of $\frac{n^{\underline{k}}}{m^k} \leq \alpha^k$, it follows that $E[P] \leq \frac{1}{2}(1 + \frac{1}{1-\alpha})$.

A.3 Analyzing Partial-Key Hashing

To reason about the costs for partial-key hashing, we need to show how insertion and probe costs depend on $S_{|L}$, the multi-set of keys. To clear up notation, we will refer to all multisets in theorems as C, C', or C''. For any multiset $C = (K, z)$, we have that $|C| = |K|$ is the number of distinct partial-key elements and $||C|| = \sum_{x\in C} z(x)$ is the total number of elements in C. We additionally extend our shorthand notation $z_x = z(x)$ to sets, i.e. $z_C = \sum_{x\in C} z_x = ||C||$. Additionally, we define $C_{2+} = x : z_x \geq 2$ to be the set of keys which are not unique in C.

Theorem 1,2 and 3 Restatements. Looking at theorem A.2.1, it's proof holds for partial-key hashing. Namely, if C is a multi-set with $||C|| = n \leq m$, and C is hashed into locations $[1, m-1]$, then the number of hash sequences such that 0 is empty is

$$(1 - \frac{n}{m})h(m, C)$$

where $h(m, C)$ is the number of ways to hash the objects into $[0, m-1]$. For a given multiset C, we have $h(m, C) = m^{|C|}$.

To generalize theorem A.2.2, we start by analyzing the form of equation (A.6) for $g(m, n, k)$. The equation consists of three parts: 1) ways to divide the set of n objects into one set with $k-1$ items and another with $n-k+1$ items, 2) ways to hash the $k-1$ and $n-k+1$ items into their arrays of size k,$m-k$, and 3) the scaling factors that say only $\frac{1}{k}, \frac{m-n-1}{m-k}$ of those leave slots 0 and k empty. It follows that when hashing a multi-set of items C, that

$$g(m, C, k) = \sum_{\substack{C'\subseteq C \\ ||C'||=k-1}} \frac{1}{k}\frac{m-n-1}{m-k}h(k, C')h(m-k, C \setminus C')$$

is the number of hash combinations that lead to a hash table with 0 empty, positions 1 to $k-1$ filled, and k empty.

Theorem A.2.3 also holds for multisets, but we need new notation to be clearer about what items are included in the new chain. We define $T_{C'}$ to be the expected chain length of a new item if all items in the multiset $C' \subseteq C$ are guaranteed to have the same hash value as the newly inserted item.

Theorem A.3.1. *Let C be the multi-set of items being hashed into the hash table and let C' be a multi-set of items such that the items in C' are fixed to have the same hash value as a newly* *inserted item for the hash table. Let $x \notin C'$, and define $C'' = C' \cup \{(x, z_x)\}$. Then*

$$P(x \in T_{C'}) = \frac{1}{m}\,\mathbb{E}[T_{C''}]$$

Proof. We start by deriving the equation for $\mathbb{E}[T_{C''}]$. Using the same reduction as in the proof of Theorem A.2.3, the number of sequences for which the a new item is hashed into a chain of length t, given that the items in C'' hash to the same location as the new item, is the same as the number of chains with 0 empty, $1, \ldots, t-1$ full, t empty, and for which the items in C'' hash to some location k between 0 and $t-1$. For a fixed chain $C^* \subset (C \setminus C'')$ with $||C^*|| = t - 1 - ||C''||$, the number of ways to hash C^* and C'' into $0, \ldots, t-1$ with the elements of C'' fixed to have the same hash location is equivalent to the number of ways to hash a new multi-set consisting of $C^* \cup (a, z_{C''})$ for an arbitrary new element a. By theorem A.2.1 for multi-sets, it follows that the number of hash combinations such that $C^* \cup C''$ are between 0 and $t-1$ and location 0 is empty is $(1 - \frac{t-1}{t})(h(t, C^*) \cdot t)$. It follows that $f(m, C, t, C'')$, the number of hash combinations that produce a chain of length t, satisfies

$$f(m, C, t, C'') = \sum_{\substack{C^* \subset (C \setminus C'') \\ ||C^*|| = t-1-||C''||}} \frac{m-n-1}{m-t} h(t, C^*) h(m-t, C \setminus (C^* \cup C''))$$

There are $m^{|C \setminus C''|}$ possible hash sequences and so it follows that

$$E[T_{C''}] = m^{-|C \setminus C''|} \sum_{t \geq 1} t f(m, C, t, C'')$$

We now derive the equation for $P(x \in T_{C'})$. The same logic again applies, with the exception that now x has a degree of freedom in its hash location. So for a fixed chain $C^* \subset (C \setminus C'')$ with $||C^*|| = t - 1 - ||C''||$, the number of ways to hash C^*, x, and C' into $0, \ldots, t-1$ with the elements of C' fixed to have the same hash location is equivalent to the number of ways to hash a new multi-set consisting of $C^* \cup \{(a, z_{C'}), (x, z_x)\}$ for an arbitrary new element a. Again by theorem A.2.1 for multi-sets, this implies the number of hash cominbations where x, C', and C^* are hashed between 0 and $t-1$ with location 0 empty is $(1 - \frac{t-1}{t})(h(t, C^*) \cdot t^2)$. It follows that

$$\begin{aligned} P(x \in T_{C'}) &= m^{-|C \setminus C'|} \sum_{t \geq 1} \sum_{\substack{C^* \subset (C \setminus C'') \\ ||C^*|| = t-1-||C''||}} \frac{m-n-1}{m-t} t \cdot h(t, C^*) h(m-k, S_{|L} \setminus C^*) \\ &= m^{-1} m^{-|C \setminus C''|} \sum_{t \geq 1} t \cdot f(m, C, t, C'') \\ &= \frac{1}{m} E[T_{C''}] \end{aligned}$$

□

Bounds on $\mathbb{E}[T_{C'}]$. Theorem A.3.1 again gives us a bridge between the probability of an item being in the same chain as a new item and the expected chain length when that item is hashed to the same location as a new item. We now use Theorem A.3.1 to prove the following bound by induction on the set of unallocated objects, i.e. $C \setminus C'$:

$$\mathbb{E}[T_{C'}] \leq z_{C'}Q_0(m, n - z_{C'}) + Q_1(m, n - z_{C'}) + \sum_{x \in (C \setminus C')_{2+}} \frac{z_x^2}{m} Q_1(m, n - z_{C'})$$

Base case: Our base case is all sets such that $C' = C$, i.e. all objects in C are guaranteed to hash to the same location as the newly inserted item. The length of T is guaranteed to be $z_{C'} + 1 \leq z_{C'}Q_0(m, 0) + Q_1(m, 0)$.
Induction Steps: Assume our induction hypothesis holds for all sets $C, C' \subset C$ with $|C \setminus C'| \leq k$. Assume now that are given sets C, C' such that $|C \setminus C'| = k + 1$, and that we wish to calculate $E[T_{C'}$. As shorthand, let $C^- = C \setminus C'$. Then

$$\begin{aligned}
\mathbb{E}[T_{C'}] &\leq (1 + z_{C'}) + \sum_{x \in C^-} z_x P(x \in T_{C'}) \\
&= (1 + z_{C'}) + \frac{1}{m} \sum_{x \in C^-} z_x E(T_{C' \cup \{(x, z_x)\}}) \\
&\leq (1 + z_{C'}) + \frac{1}{m} \sum_{x \in C^-} z_x \big[(z_x + z_{C'})Q_0(m, n - z_{C'} - z_x) + Q_1(m, n - z_{c'} - z_x) \\
&\quad + \sum_{y \in C^-, y \neq x} \frac{z_y^2}{m} Q_1(m, n - z_{C'} - z_x)\big] \\
&\leq (1 + z_{C'}) + \frac{n - z_{C'}}{m} z_{C'} Q_0(m, n - z_{C'} - 1) + \sum_{x \in C^-} \frac{z_x^2 + z_x}{m} Q_0(m, n - z_{C'} - 1) \\
&\quad + \frac{n - z_{C'}}{m} Q_1(m, n - z_{C'} - 1) + \sum_{x \in C^-} \frac{z_x^2}{m} \frac{n - z_{C'}}{m} Q_1(m, n - z_{C'} - 1) \\
&= z_{C'}(1 + \frac{n - z_{C'}}{m} Q_0(m, n - z_{C'} - 1)) && \text{(A.7)} \\
&\quad + 1 + \frac{n - z_{C'}}{m} Q_0(m, n - z_c - 1) + \frac{n - z_{C'}}{m} Q_1(m, n - z_{C'} - 1) && \text{(A.8)} \\
&\quad + \sum_{x \in C^-_{2+}} \frac{z_x^2}{m} (Q_0(m, n - z_c - 1) + \frac{n - z_{C'}}{m} Q_1(m, n - z_{C'} - 1)) && \text{(A.9)}
\end{aligned}$$

Now, using the rules $\frac{n}{m} Q_1(m, n-1) = Q_1(n, m) - Q_0(n, m)$ and $\frac{n}{m} Q_0(m, n-1) = Q_0(m, n) - 1$ we can reduce equations (A.7),(A.8), and (A.9) respectively to $z_{C'}Q_0(m, n - z_{C'}), Q_1(m, n - z_{C'})$, and $\sum_{x \in (C \setminus C')_{2+}} \frac{z_x^2}{m} Q_1(m, n - z_{C'})$ respectively. This gives the required result on $\mathbb{E}[T_{C'}]$.

It follows for a query on a new item k such that $L(k) = y$, that it has a bound on its expected chain length of

$$E[T] \leq z_y Q_0(m, n - z_y) + Q_1(m, n - z_y) + \sum_{x \neq y \in C} \frac{z_x^2}{m} Q_1(m, n - z_y)$$

Connecting $\mathbb{E}[T]$ to $\mathbb{E}[P']$. Depending on whether $z_y = 0$, the connection between chain length and probe length changes. For items which match no other partial-keys in the dataset, we can use the same uniformity assumptions as before to recover that $\mathbb{E}[P'] = \frac{1}{2} + \frac{1}{2}\mathbb{E}[T]$. For items which match at least one partial-key, this uniformity assumption does not hold, and indeed items with larger c_y values are more likely to be hashed towards the beginning of their chain. Thus, we use that $P \leq T$ to say $\mathbb{E}[P] \leq \mathbb{E}[T]$, and note this is a loose bound when z_y is small. The result, overall, is equation A.3, restated here for convenience.

$$\mathbb{E}[P'] \leq \begin{cases} \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2} + \sum_{x \neq y} \frac{z_x^2}{m(1-\alpha)^2}) & \text{if } z_y = 0 \\ \frac{z_y}{1-\alpha} + \frac{1}{(1-\alpha)^2} + \sum_{x \neq y} \frac{z_x^2}{m(1-\alpha)^2} & \text{if } z_y > 0 \end{cases}$$

Average query cost for an existing key. The average cost to query a key in linear probing is unaffected by the order in which keys are inserted. Thus, we may assume any insertion order we desire in order to ease the analysis of the average number of probes for existing keys.

Because our analysis in the prior section created looser bounds for $z_y > 0$, namely in that there was no uniformity assumption on where items were in a chain, we add all duplicate keys first and then all non-duplicate keys after. Expanding this out, we have

$$\begin{aligned}
\mathbb{E}[P] &\leq \frac{1}{n}\Big(\sum_{x \in C_{2+}} \frac{z_x^2}{2} Q_0(m,d) + \sum_{i=0}^{d-1}\big[Q_1(m,i) + \sum_{x \in C_{2+}} \frac{z_x^2}{m} Q_1(m,i)\big]\Big) \\
&\quad + \frac{1}{2n}\big[\sum_{i=d}^{n-1}(1 + Q_1(m,i) + \sum_{x \in C_{2+}} \frac{z_x^2}{m} Q_1(m,i))\big] \\
&\leq \frac{n-d}{2n} + \frac{1}{2}Q_0(m,n-1) + \frac{c+d}{2n}Q_0(m,d-1) + \frac{c}{2m}(Q_0(m,n-1) + \frac{d}{n}Q_0(m,d-1)) \\
&\approx \frac{1}{2}(1 + \frac{1}{1-\alpha}) + \frac{c}{n} + \frac{c}{m}\frac{1}{1-\alpha} \\
&\leq (\frac{1}{2} + \frac{c}{n})(1 + \frac{1}{1-\alpha})
\end{aligned}$$

Here we use that $\alpha_d = \frac{d}{m} \approx 0$ so that $\frac{1}{1-\alpha_d} \approx 1$. Because we are interested in the case that duplicate items are rare, this is a very good approximation.

Random Data. To go from fixed data to random data, we condition via Adam's law. For querying for a missing key, we first rewrite the expectation as

$$\begin{aligned}
\mathbb{E}[P'|y, S_{|L}] &\leq \frac{1}{2} + \mathbb{1}_{z_y \neq 0}(\frac{z_y}{1-\alpha} - \frac{1}{2}) + (\frac{1}{2} + \frac{1}{2}\mathbb{1}_{z_y \neq 0})\frac{1}{(1-\alpha)^2} + (\frac{1}{2} + \frac{1}{2}\mathbb{1}_{z_y \neq 0})\sum_x \frac{z_x^2}{m}\frac{1}{(1-\alpha)^2} \\
&\leq \frac{1}{2}(1 - \frac{1}{(1-\alpha)^2}) + \frac{\mathbb{1}_{z_y \neq 0}\, z_y}{1-\alpha} + \frac{\mathbb{1}_{z_y \neq 0}}{(1-\alpha)^2} + \sum_x \frac{z_x^2}{m}\frac{1}{(1-\alpha)^2}
\end{aligned}$$

Then using that $E[\mathbb{1}_{z_y \neq 0}] \leq \mathbb{E}[z_y] = \mathbb{E}[\mathbb{1}_{z_y \neq 0}\, z_y]$, that $\mathbb{E}[z_y] \leq n2^{-H_2(L(X))}$ by the union bound, and that $E[\sum_{x \in S_L} z_x^2] = n^2 2^{-H_2}$, we have

$$\mathbb{E}[P'] \leq \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2}) + \frac{n2^{-H_2(L(X))}}{1-\alpha} + \frac{n2^{-H_2 L(X))}}{2(1-\alpha)^2} + \sum_x \frac{\alpha n 2^{-H_2(L(X))}}{(1-\alpha)^2}$$

$$\leq \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2}) + n2^{-H_2(L(X))}\frac{3}{2(1-\alpha)^2}$$

More straightforwardly,

$$\mathbb{E}[P] \leq \frac{1}{2}(1 + \frac{1}{1-\alpha}) + n2^{-H_2(L(X))}(1 + \frac{1}{1-\alpha})$$

This concludes the analysis of linear probing, proving the initial equations given in Chapter 3.

Appendix B

Appendix: Stacked Filters

B.1 Proof of Size Concentration Bounds

Theorem B.1.1. *Let s' be the size of a filter using Algorithm 3, and assume that a Stacked Filter of T_L layers false positive rate α_i for each layer $i \in \{1, \dots, T_L\}$. Let $\alpha_{max} = max_i \alpha_i \leq \frac{1}{2}$ and $\alpha_{min} = \min_i \alpha_i$. Then*

$$\mathbb{P}(|s' - E[s']| \geq kE[s']) \leq$$

$$\frac{1}{|P|} \cdot \frac{1}{k^2} \cdot (\frac{s(\alpha_{min})(1-\alpha_{min})}{s(\alpha_{max})(1-\alpha_{max})})^2 \cdot \frac{(1+\frac{|N_f|}{|P|})\alpha_{max}}{(1+\frac{|N_f|}{|P|}\alpha_{min})^2}$$

Proof. The result is an application of Chebyshev's inequality. Start by letting $X_N^{n,j}, j \in \{1, \dots, |N_f|\}$ denote the event that item j of the set N_f made it to negative layer n in construction. Similarly, $X_P^{n,j}$ denote the event positive item j made it to positive layer n. Since s', s are in bits per positive element, we have

$$s' = \frac{1}{|P|} \sum_{j=1}^{|P|} \sum_{i=1}^{(T_L+1)/2} s(\alpha_{2i-1}) X_P^{i,j} + \sum_{j=1}^{|N_f|} \sum_{i=1}^{(T_L-1)/2} s(\alpha_{2i}) X_N^{i,j}$$

By using α_{max} in the size equation and α_{min} for the conditional chance of making it to the next layer, we have $E[s'] \geq s(\alpha_{max}) \frac{1+\frac{|N_f|}{|P|}\alpha_{min}}{1-\alpha_{min}}$.

For the variance part, we have that $X_{C_1}^{m,j} \perp\!\!\!\perp X_{C_2}^{m',j'}$ unless $C_1 = C_2$ and $j = j'$ by our method of construction. Thus

$$\text{Var}[s'] \leq (\frac{s(\alpha_{min})^2}{|P|})(\text{Var}[\sum_{i=1}^{\frac{T_L+1}{2}} X_P^{i,1}] + \frac{|N_f|}{|P|}\text{Var}[\sum_{i=1}^{\frac{T_L-1}{2}} X_N^{i,1}])$$

To break down the summation, we use Eve's law. Letting $\mathbb{F}_N^n$ be the filtration containing all events up to negative layer n, we have

$$\begin{aligned}\text{Var}[\sum_{i=1}^{n} X_N^{i,1}] &= E[\text{Var}[\sum_{i=1}^{n} X_N^{i,1} \mid \mathbb{F}_N^{n-1}] + \text{Var}[E[\sum_{i=1}^{n} X_N^{i,1}] \mid \mathbb{F}_N^{n-1}] \\ &\leq (1-\alpha_{max})\alpha_{max}^n + \text{Var}[\sum_{i=1}^{n-2} X_N^{i,1} + (1+\alpha_{max})X_N^{n-1,1}]\end{aligned}$$

From here, one can use Eve's law recursively to expand the expression on the right until reaching an expression only involving X_N^0. Evaluating this gives:

$$\begin{aligned}\text{Var}[\sum_{i=1}^{n} X_N^n] &\leq \sum_{j=1}^{n} (\sum_{i=0}^{j} \alpha^i)^2 \cdot \alpha_{max}^{n-j} \cdot (\alpha_{max})(1-\alpha_{max}) \\ &\leq \frac{\alpha_{max}}{(1-\alpha_{max})^2}\end{aligned}$$

We note that this makes intuitive sense as it is less variance than a geometric series using success parameter $(1-\alpha_{max})$. A similar result holds for $\text{Var}[\sum X_P^{n,1}]$, and combining these two results with the variance expression above, the expected value expression above, and using Chebyshev's inequality gives the desired result. □

B.2 Derivation of Computational Costs

Construction. For both positives and frequently queried negatives, the construction algorithm is best analyzed in pairs, wherein the same number of elements are inserted at one layer and queried against the next. For positives, every element is inserted into L_1 and checked against L_2. The false positives from L_2 are then inserted into L_3 and checked against L_4, and so on. The total cost, in terms of base filter operations, is $|P|(c_i + c_q) + |P|\alpha_2(c_i + c_q) + |P|\alpha_2\alpha_4(c_i + c_q) + \ldots$. In more concise notation, this is

$$|P|(c_i + c_q)(\sum_{i=0}^{\frac{T_L-1}{2}-1} \prod_{j=0}^{i} \alpha_{2i}) + |P|c_i \prod_{j=0}^{\frac{T_L-1}{2}} \alpha_{2i}$$

where the final term comes from the last layer insertions.

For negative layers, the analysis is similar, but the paired layers are instead 2 and 3, 4 and 5, and so on, with frequently queried negatives having the first layer unpaired instead of the last. The number of operations to insert negatives is

$$|N_f|c_q + |N_f|(c_i + c_q) \sum_{i=0}^{\frac{T_L-1}{2}} \prod_{j=1}^{i} \alpha_{2j-1}$$

The total construction cost is the sum of the operations for positive and frequently queried negative elements.

In the case that the α values are all equal, the total cost can be bounded using geometric series by

$$|P|(c_i + c_q)\frac{1}{1-\alpha} + |N|c_q + |N|(c_i + c_q)\frac{\alpha}{1-\alpha}$$

In this case, comparing against the cost for a single base filter of $c_i \cdot |P|$, the overhead is seen to be the cost of querying a base filter for both every positive and frequently queried negative element once, plus a small overhead from stacking.

Query Costs. The analysis of querying a Stacked Filter for a positive or frequently queried negative is almost exactly the same as the analysis for constructing a Stacked Filter, except that inserts are replaced with queries. For instance, when querying a positive, it queries L_1 and L_2 with certainty, L_3 and L_4 with probability α_2, L_3 and L_4 with probability $\alpha_2\alpha_4$ and so on. The resulting equations for positive and frequently queried negative elements are

$$c'_{q,P} = 2c_q(\sum_{i=0}^{\frac{T_L-3}{2}} \prod_{j=0}^{i} \alpha_{2i}) + c_q \prod_{j=0}^{\frac{T_L-1}{2}} \alpha_{2i}$$
$$c'_{q,N_f} = c_q + 2c_q \sum_{i=0}^{\frac{T_L-1}{2}} \prod_{j=1}^{i} \alpha_{2j-1}$$

For an infrequently queried negative element, it can be rejected by both positive and negative layers. Thus, the probability of it reaching layer i is the product of the false positive rates for layers $1, \ldots, i-1$

$$c'_{q,N_i} = c_q(\sum_{i=0}^{T_L} \prod_{j=0}^{i} \alpha_i)$$

Letting the FPRs at each layer be equal and using geometric series bounds, we have:

$$c'_{q,P} \leq \frac{2}{1-\alpha}c_q, \quad c_{q,N_f} \leq \frac{1+\alpha}{1-\alpha}c_q, \quad c'_{q,N_i} \leq \frac{1}{1-\alpha}c_q$$

B.3 Proofs

B.3.1 Proof of Quasiconvexity

Theorem B.3.1. *The size function* $s(\alpha) = \frac{-\log_2(\alpha)}{f} * (\frac{1}{1-\alpha} + \frac{|N_f|}{|P|}\frac{\alpha}{1-\alpha})$ *is quasiconvex on (0,1) for* $f > 0$.

Proof. For functions of a single variable, a sufficient condition for quasiconvexity is that s' starts negative, has at most one root, and is then positive after (in the case a root exists). We use this to prove the quasiconvexity of s. For notational convenience, we will replace $\frac{|N_f|}{|P|}$ with N, and note $N > 0$.

We have

$$s'(\alpha) = \frac{N\alpha^2 + (1-N)\alpha - (N+1)\alpha \ln \alpha - 1}{((1-\alpha)^2 \alpha \ln 2)}$$

Let f be the numerator of s', i.e. $N\alpha^2 + (1-N)\alpha - (N+1)\alpha \ln \alpha - 1)$, and note the denominator is positive for $\alpha \in (0,1)$. It follows that $\text{SIGN}(s') = \text{SIGN}(f)$, and therefore it is enough to show f starts negative and either has no roots, or has one root and is positive for all values after its root.

First, as $\alpha \to 0$, $f(\alpha) \to -1$, so f starts negative. Second, we note the functional forms of f', f'':

$$f'(\alpha) = 2N(\alpha - 1) - (N+1)\ln\alpha$$
$$f''(\alpha) = 2N - \frac{N+1}{\alpha}$$

By inspection, $f''(\alpha)$ has one root in $\mathbb{R}$, and so by Rolle's Theorem, $f'(\alpha)$ has at most two roots in $\mathbb{R}$. $f'(1) = 0$, so it has at most one other root. Since $\lim_{\alpha\to 0} f'(\alpha) = \infty$, then if the other root is after 1, f is monotonically increasing on $(0,1)$ and we are done. Otherwise, f' has a single root $\alpha_1 \in (0,1)$. It follows that f cannot have a root in $[\alpha_1, 1)$ since $f(1) = 0$ and $f(\alpha_2) = 0$ for $\alpha_2 \in [\alpha_1, 1)$ would imply the existence of another root for f'.

Since $f'(\alpha) > 0 : \forall\alpha \in (0, \alpha_1)$, it follows that if f has a root in $\alpha^* \in (0, \alpha_1)$, then $f(\alpha) > 0 : \forall\alpha \in (\alpha^*, 1)$. This is what we set out to show and so s is quasiconvex. ☐

Theorem B.3.2. *The function* $s(\alpha) = \frac{-\log_2(\alpha)+c}{f} * (\frac{1}{1-\alpha} + \frac{|N_f|}{|P|}\frac{\alpha}{1-\alpha})$ *is quasiconvex on (0,1) when* $c \geq 0, f > 0$.

Proof. As before let $N = \frac{|N_k|}{|P|}$. Take the derivative of s, note the denominator is positive, and form f_c from the numerator of s', which in this case is $N\alpha^2 + (1-N)\alpha - (N+1)\alpha\ln\alpha - 1 + (N+1)c\alpha\ln 2$. As in the proof of Theorem B.3.1, $\text{SIGN}(f_c) = \text{SIGN}(s')$. Additionally, note that f_c is equal to f from the previous problem plus an additional positive term: $(N+1)c\alpha\ln 2$.

In the case that f is an increasing function on $(0,1)$, then so is f_c and we are done. Otherwise, let f' have a root α_1. Then on $(0, \alpha_1)$ both f and f_c are increasing functions and so have at most one root. On $(\alpha_1, 1)$, $f > 0$ and so $f_c > 0$; thus both have 0 roots in $(\alpha_1, 1)$. Finally, we note that $lim_{\alpha\to 0} f_c(\alpha) = -1$ and $f_c(1) = (N+1) * c * \ln 2 > 0$. Thus f_c starts negative, has exactly one root, and is then positive. The same is therefore true of s' and so $s(\alpha)$ is quasiconvex. ☐

B.3.2 Sweep over N Proof

Theorem. *Given an oracle returning the optimal EFPR for a given set* N_f *satisfying all constraints, finding the optimal EFPR across all values of* $|N_f|$ *to within* ϵ *requires* $O(\frac{1}{\epsilon})$ *calls to the oracle.*

Proof. For $i \in \{1, 2\}$, let N_i be the set of n_i most heavily queried elements, let $\psi_i = P(x \in N_i | x \in N)$, let $EFPR_i^*$ be the best EFPR using N_i satisfying all constraints, and let $P_i(x)$ be shorthand for the chance x is a false positive using the Stacked Filter giving $EFPR_i^*$. Then

$$\begin{aligned}
EFPR_2^* &= \psi_2 P_2(x|x \in N_f) + (1-\psi_2)P_2(x|x \in N_u) \\
&= \psi_1 P_2(x|x \in N_f) + (1-\psi_1)P_2(x|x \in N_u) \\
&\quad + (\psi_2 - \psi_1)(P_2(x|x \in N_f) - P_2(x|x \in N_u)) \\
&\geq EFPR_1^* - (\psi_2 - \psi_1)P_2(x|x \in N_u) \\
&\geq EFPR_1^* - (\psi_2 - \psi_1)
\end{aligned}$$

As a result, we can start at $|N_f| = 0$ elements and scan the most frequently queried elements in order. When more than ϵ difference exists in the ψ values between the last call to the oracle and the current N_f, then we call the oracle again. By the above statements, this produces an EFPR within an ϵ amount of the best EFPR possible across all possible sets for N_f. ☐

Both Algorithm 7 and the algorithm for optimizing continuous FPR Stacked Filters use less optimization subroutines than the proof above, although the number is still $O(\frac{1}{\epsilon})$. The change comes modifying the final line of the proof, using the fact that if $EFPR_2^* \leq \alpha$, where α is the FPR of a 1-layer Stacked Filter, then $P_2(x|x \in N_u) \leq \frac{\alpha}{1-\psi_2}$.

B.3.3 Proof of Theorem 4.7.1

We restate Theorem 4.7.1 here for convenience:

Theorem. *Assume the equation for the size of a base filter in bits per positive element is of the form* $s(\alpha_i) = \frac{-\log_2(\alpha_i)+c}{f}$*. Let the positive set have size* $|P|$*, let the distribution of our negative queries be* D*, and let* α *be a desired EFPR. If there exists any set* N_f*,* $\psi = \mathbb{P}_D(x \in N_f | x \in N)$*, and* $k \leq \psi$ *such that*

$$\frac{|N_f|}{|P|} \leq \frac{\ln \frac{1}{1-k}}{\ln \frac{1-k}{\alpha} + c} \cdot \frac{1-k-\alpha}{\alpha} - 1$$

then a Stacked Filter (optimized using Section 4.5 and given access to any N_f *satisfying the constraint) achieves the EFPR* α *using fewer bits than a query-agnostic filter.*

Proof. We limit ourselves to Stacked Filters which have an equal FPR α_L at each layer. In this case, we have that the EFPR of the filter is

$$\psi \alpha_L^{T_L - 1/2} + (1-\psi)[(1-\alpha_L) * (\alpha_L + \alpha_L^3 + ... + \alpha^{T_L - 2}) + \alpha^{T_L}]$$

For some N, $T_L \geq N$ implies this is less than $(1-\psi)\alpha_L$. Thus, a Stacked Filter of length N and $\alpha_L \leq \frac{\alpha}{1-\psi}$ has an EFPR less than α.

Our goal is then to show that there exists a Stacked Filter with $\alpha_L \leq \frac{\alpha}{1-\psi}$ which has size less than $\frac{-\ln(\alpha)+c}{f}$. From Section 4, we have that the size of a Stacked Filter is bounded above by

$$s(\alpha_L)\left(\frac{1}{1-\alpha_L} + \frac{|N_f|}{|P|}\frac{\alpha_L}{1-\alpha_L}\right)$$

Plugging in our equations for the size of a base filter, a Stacked Filter is better than a traditional filter if $0 \leq \alpha_L \leq \frac{\alpha}{1-\psi}$ and

$$\frac{-\ln \alpha_L + c}{f \ln 2}\left(\frac{1}{1-\alpha_L} + \frac{|N_f|}{|P|}\frac{\alpha_L}{1-\alpha_L}\right) \leq \frac{-\ln \alpha + c}{f \ln 2} \tag{B.1}$$

We now let $\alpha_L = \frac{\alpha}{1-k}$, and our EFPR equation gives the restraint $0 \leq k \leq \psi$. We further break apart our equation for size into two parts: our goal will be to show that $\frac{1}{1-\alpha_L} + \frac{|N_f|}{|P|}\frac{\alpha_L}{1-\alpha_L} \leq 1 + b$, and that $(1+b)\frac{-\ln \alpha_L + c}{f \ln 2} \leq \frac{-\ln \alpha + c}{f \ln 2}$. If both equations are satisfied, then (B.1) is satisfied as well.

Part 1: $(1+b)\frac{-\ln \alpha_L + c}{f \ln 2} \leq \frac{-\ln \alpha + c}{f \ln 2}$: Plugging in $\alpha_L = \frac{\alpha}{1-k}$ and solving this inequaliity gives: $b \leq \frac{-\ln 1-k}{-\ln \frac{\alpha}{1-k} + c}$.

Part 2: $\frac{1}{1-\alpha_L} + \frac{|N_f|}{|P|}\frac{\alpha_L}{1-\alpha_L} \leq 1 + b$: Rearranging, we get $\frac{|N_f|}{|P|} \leq \frac{b}{\alpha_L} - 1 - b$. We now set $b = \frac{\ln 1-k}{\ln \frac{\alpha}{1-k}}$,

Algorithm 7 Optimization of fingerprint-based filters

```
Input: $\psi_{dist}$: a function translating $|N_f|$ into a $\psi$ value
Input: $s$: a size constraint in bits/element
Input: $\epsilon$: a slack from optimum
 1: FPR_1L = FPR_SINGLE_LAYER(size)
 2: bestFPR, bestSetup = FPR_1_Layer, $\{\alpha_{1L}\}$
    // remove constant load factor $f$.
    // Size eq. becomes $s(\alpha) = -\log_2(\alpha) + c$
 3: $s' = \frac{s}{f}$
 4: lastPsi = 0
 5: for $|N_f| = \lceil \frac{|P|}{1000} \rceil$ to $|N_{samp}|$ do
 6:     newPsi = $\psi_{dist}(N_f)$
 7:     if $newPsi - lastPsi \cdot \min(1, \frac{FPR\_1L}{1-newPsi}) \geq \frac{\epsilon}{2}$ then
 8:         FPR, setup = optNFixed(s', newPsi, $\frac{|N_f|}{|P|}$, $\frac{\epsilon}{2}$)
 9:         lastPsi = newPsi
10:         if FPR < bestFPR then
11:             bestFPR, bestSetup = FPR, setup
12: return bestFPR, bestSetup
```

satisfying the inequality above, and plug in $\frac{\alpha}{1-k}$ for α_L. The resulting equation is

$$\frac{|N_f|}{|P|} \leq \frac{\ln \frac{1}{1-k}}{\ln \frac{1-k}{\alpha} + c} \cdot \frac{1-k-\alpha}{\alpha} - 1$$

as desired, with the constraint that $0 \leq k \leq \psi$.

To see that the constructed Stacked Filter achieves this goal, note that as $\epsilon \to 0$, the optimization schemes of Section 5 approach the optimal filter using an equal FPR at all layers. □

B.4 Optimizing Integer length Fingerprint Filters

The optimization algorithms for integer length fingerprint Stacked Filters work similarly to continuous FPR filters. One routine loops through possible values of $|N_f|$, always choosing greedily the most heavily queried elements, and calls another routine to optimize a Stacked Filter for a given N_f and ψ. The first routine does so in a way so as to make sure that the skipped values can produce no more than an ϵ better solution, as proven in Theorem 4.5.1.

B.4.1 Breadth First Search on Stacked Filters

Algorithms 8 and 9 deal with the optimization of an integer fingerprint filter given N_f and ψ. The description of the algorithm is split between the pseudocode given and various steps described in the text here. For integer-length fingerprint filters, the optimization algorithm uses breadth first search on the discrete options available, and its goal is to find a solution within ϵ of the best solution possible. The breadth first search starts with an empty stack, and at each step expands upon the current stack. It does so both by 1) finishing the current stack with a positive layer of maximum size (lines 6-9 of Algorithm 8), which puts no new options onto the BFS queue and 2) looping over the possible fingerprint lengths for 1 positive and 1 negative layer (lines 11-13 of Algorithm 8), and adding each of these deeper stacks onto the BFS queue.

Algorithm 8 optNFixed

```
Input: $\psi, s, \frac{|N_f|}{|P|}, \epsilon$
    // construction values below correspond to order they appear in Algorithm 9
 1: startObject = OptObj(s,$\psi$, $\frac{|N_f|}{|P|}$, 1.0, 0.0)
 2: queue = {startObject}
 3: bestFPR, bestSetup = FPR_1_Layer, {$\alpha_{1L}$}
    // begin breadth first search
 4: while !queue.empty() do
 5:     curObj = queue.pop()
        // calculate FPR with this as final layer and check if best
 6:     addedFPR = $2^{c-\lfloor \text{curObj.s} \rfloor}$ × curObj.prop
 7:     if curObj.FPR + addedFPR < bestFPR then
 8:         bestFPR = curObj.FPR + addedFPR
 9:         bestSetup = curObj.fingerprints +{⌊curObj.s⌋}
        // check if prop. × FPR final layer less than $\epsilon$
        // since best FPR is $\geq 0$, implies best expansion within $\epsilon$
10:     if addedFPR < $\epsilon$ then continue
        // perform breadth first search. See text for bounds
11:     for i = f1_min; i ≤ ⌊curObj.s⌋; i++ do
12:         for j=c+1; j < f2_max; j++ do
13:             queue.push(createNewOptObj(f1,f2,curObj))
14: return bestFPR, bestSetup
```

A Brute Force Approach. As there are a discrete number of options for fingerprints at each layer (given our bounds on s), the BFS will build all possible 1 layer stacks, then all possible 3 layer stacks, then all possible 5 layer stacks, etc. Taking the best FPR solution of each of these would result in the exact optimal solution of the best Stacked Filter with 1 layers, 3 layers, 5 layers, etc. However, the number of possible Stacked Filters grows exponentially with the number of layers and so this is infeasibly slow even for just 7 or 9 layers. Additionally there isn't any guarantee about how close the found solution is to the optimal.

From Brute Force to ϵ-approximation. A more efficient solution is close each search path (i.e. a partially built stack) by building a final positive layer in a way that preserves the performance of BFS. In particular, we do so when using a final positive layer produces a solution within ϵ EFPR of the best possible Stacked Filter starting with the same initial layers as the current partially built stack. Then, by aggregating the best seen result across all search paths, the overall algorithm finds a Stacked Filter within ϵ EFPR of the optimal.

Stopping Condition. A search path is stopped when the proportion of negative queries which reach the end of the partial stack multiplied by the FPR of a single layer built using all available space budget is less than epsilon. As an example, assume we have a four layer stack each with FPR 0.01, that the initial $\psi = 0.5$, that $\epsilon = 10^{-5}$, and that using all space left for the filter on a single positive layer gives a final layer with FPR 0.01. The proportion of queries which will reach the newly built 5th layer is $0.5 * 0.01^2 + 0.5 \cdot 0.01^4 \approx 5 \cdot 10^{-4}$, where the first term is the proportion of queries from N_f and the second is the proportion from N_i. The best hypothetical Stacked Filter rejects all these queries, resulting in no false positives from layer 5 onwards; a single layer stack would have an approximate EFPR of $10^{-2} \cdot 5 \cdot 10^{-4} = 5 \cdot 10^{-6}$ from layer 5 onwards, which is less than ϵ different from rejecting all queries. Thus it is suitably close to the best possible Stacked Filter starting with four layers of FPR = 0.01, and so we choose the one layer stack and terminate

Algorithm 9 createNewOptObj

```
Input: $f_1, f_2$: fingerprint sizes of new positive and negative layer
Input: optObj: prior optimization object
 1: $\alpha_1, \alpha_2 = 2^{c-f_1}, 2^{c-f_2}$
 2: newOpt.s = (optObj.s - $f_1$ - objObj.$\frac{|N_f|}{|P|} \times \alpha_1 \times f_2)/\alpha_2$;
 3: newOpt.$\psi = \frac{optObj.\psi}{optObj.\psi + (1-optObj.\psi)\times\alpha_2}$
 4: newOpt.$\frac{|N_f|}{|P|}$ = optObj.$\frac{|N_f|}{|P|} \times \frac{\alpha_1}{\alpha_2}$
    // prop is the proportion of negative queries which reach the end of the stack
 5: newOpt.prop = optObj.prop $\times\alpha_1 \times$ (obtObj.$\psi + (1 -$ optObj.$\psi) \times \alpha_2)$
 6: newOpt.FPR = optObj.FPR + $(1 -$ optObj.$\psi \times \alpha_1 \times (1 - \alpha_2))$
 7: newOpt.fingerprints = optObj.fingerprints $+\{f_1, f_2\}$
 8: return newOpt
```

the search path. The stopping condition is lines 6-9 of Algorithm 8.

Expanding the Search. To perform the BFS, we need to figure out which limited paths need to be searched to produce an optimal solution. One obvious constraint is that all search paths should be feasible; i.e. we should only choose fingerprint lengths for each layer that do not go over the allowed filter size. A second obvious constraint is that layers should have FPR less than 1, which gives that the maximum FPR a filter has will be $\frac{1}{2}$.

A third set of constraints comes from the following bounds, which is derived from the fact that the derivative of size with respect to α_1 should never be positive (as higher α_1 values always lead to worse EFPR, it makes no sense to use α_1 values which have worse EFPR and size). The second equation is just a rewritten version of the first.

$$\alpha_1 < \frac{|P|}{|N_f|} \cdot \frac{1}{\ln 2} \cdot \frac{1}{-\log_2 \alpha_2 + c}$$
$$\alpha_2 > 2^{-\frac{|P|}{|N_f|}\frac{1}{\ln 2}\frac{1}{\alpha_1}+c}$$

The bounds on α_1 and α_2 then become the following bounds for fingerprint sizes, where f_1, f_2 are the integer-valued fingerprint sizes for elements in L_1 and L_2:

$$f_1 > \lfloor -\log_2 \frac{|P|}{|N_f|} \frac{1}{1+c} \frac{1}{\ln 2} \rfloor \tag{B.2}$$
$$f_2 < \lceil \frac{|P|}{|N|} \frac{1}{\ln 2} 2^{f_1 - c} \rceil \tag{B.3}$$

In the first equation we plugged in $\alpha_2 \leq \frac{1}{2}$ since we choose α_1 before α_2 in Algorithm 8. The ceiling and floor functions comes from the fact that the problem is discrete.

Recursive Nature of BFS. One of the key insights of the BFS algorithm is that optimizing a Stacked Filter starting at level 3 (i.e. the first two levels are decided already) is equivalent to optimizing a Stacked Filter at level 1. In particular, because the problem of designing an optimal stack from layer 3 onwards is identical to designing a stack from layer 1, the bounds above on f_1 and f_2 apply each time we look into building 2 new layers, (using an updated value of $\frac{|N_f|}{|P|}$ which represents the expected sizes of $|N_f|$ and $|P|$ reaching this layer).

Algorithm 9 updates all the parameters for the next layer. That is, it tracks parameters ψ, s, and

$\frac{|N_f|}{|P|}$ after going through 2 layers, where these values represent their values conditioned on making it to the current layer. For instance, the new ψ value represents what proportion of queries reaching the current layer are from N_f vs. from N_i. The parameter 'prop' tells what portion of negative queries make it to this layer, and the FPR and fingerprints parameters keep track of the already accrued false positives from negative layers and the chosen fingerprint sizes so far.

B.5 3-Layer Optimization of Adaptive Stacked Filters

This section details the process of optimizing a 3-layer ASF. As input the optimization has a number of queries the filter is expected to last, denoted by F_{TTL} as well as a size budget s. We start by searching over a range of values for N_o, with the search performed logarithmically over values between 1000 and F_{TTL}, i.e. each value of N_o checked is 1.4 times as big as the previous value.

The value of N_o determines the expected values of ψ and $|N_f|$, which are:

$$E[\psi] = \sum_{x \in N_{samp}} f(x)(1-(1-f(x))^{|N_o|})$$
$$E[N_f] = (\ell \cdot |N_o|) + \sum_{x \in N_{samp}} (1-(1-f(x))^{|N_o|})$$

Recall that ℓ is the estimate of what portion of queries fall on values outside our histogram. The equations pessimistically assume in the second equation that all elements of N_f which were not present in the histogram sample will never again be queried, i.e. they are one-off queries. If the universe of elements is not extremely large or if the query pattern of the elements not present in the histogram is skewed, then the true $E(\psi)$ may be higher, and the true $E(N_f)$ may be smaller.

If the size of N_{samp} is large, then calculating exactly $E[\psi]$ and $E[N_f]$ can be expensive. Thus, when dataset sizes are larger than 1 million elements, we sample uniformly with replacement 100,000 values of N_{samp} and estimate $E[\psi]$ and $E[N_f]$ by:

$$E[\psi] = (1-\ell) - \frac{|N_{samp}|}{100,000} \sum_{i=1}^{100,000} f(x_i) \cdot (1-f(x_i)^{|N_o|}$$
$$E[|N_f|] = (\ell \cdot N_o) + \frac{|N_{samp}|}{100,000} (\sum_{i=1}^{100,000} 1-(1-f(x_i))^{N_o})$$

In the above estimations, we make use of the fact that $\sum_{x \in N_{samp}} f(x) = \ell$, and hence we only estimate the $-f(x) \cdot (1-f(x))^{N_o}$ term. This considerably reduces the variance of the estimate.

The optimization goal is to optimize the weighted sum of the EFPR using only L_1 for the first N_o queries and the EFPR of the ASF using all 3 layers on the rest of the queries. To do this, we use discrete search on the FPRs of each layers for both continuous FPR filters and integer-length fingerprint filters. For integer-length fingerprint filters we check all fingerprint sizes on each layer which are between $[c+1$ and $c+16]$ so that the FPRs are between $\frac{1}{2}$ and 2^{-16}, and return the best configuration satisfying the size constraint. For the continuous FPR filters, we similarly use size bounds which provide FPRs of $\frac{1}{2}$ and 2^{-16} on each layer, and then perform grid search where we check sizes that are 0.1 bits per element apart. As before, we return the best configuration that satisfies the size constraint.

Figures B.1 and B.2 display the optimization times and FPRs for ASFs on the synethetic integer

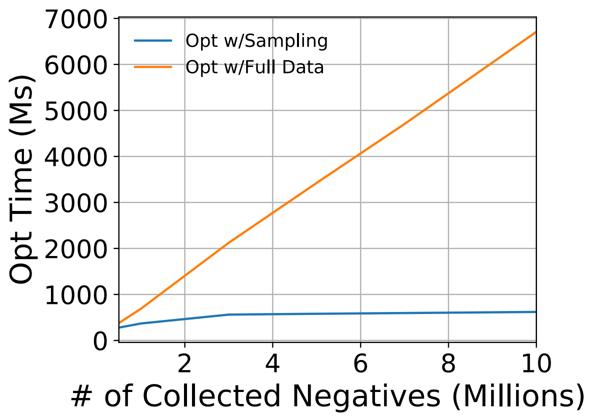


Figure B.1: full sample

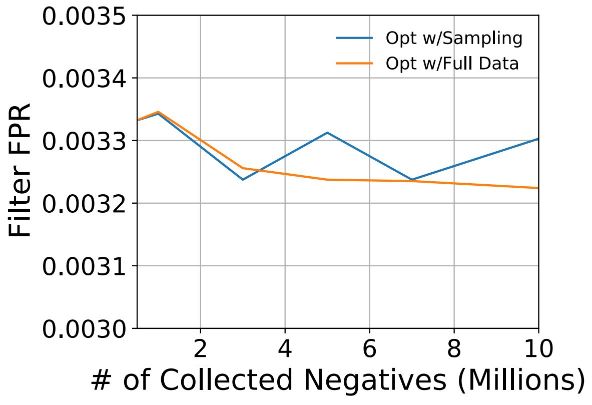


Figure B.2: w/ subsampling

dataset with and without sampling. The dataset has 1 million positive values, 100 million negative values, and a zipf of 1. The number of negative values in N_{samp} is the dependent variable. In general, for small datasets the times are comparable and execute in sub-second time. As the dataset size grows, the sampling based approach becomes significantly faster while maintaining similar accuracy.

Appendix C

Appendix: Column Sketches

C.1 Multi-Column Sketches

Overview. Multi-Column Sketches take as input one or more columns for which the resulting Column Sketch will be non-order preserving, and (optionally) a single column for which order is preserved. We expect Multi-Column Sketches to be of use in combining categorical variables that are queried frequently together, with perhaps a single numerical attribute. Like single column Column Sketches, we expect Multi-Column Sketches to work when the domain of the Column Sketch is larger than 256; however, the size of the domain is measured in terms of the possible joint values of the column, and so Multi-Column Sketches are useful in combining various categorical attributes which individually require less than 9 bits. Because a single encoding cannot support order on multiple attributes, we do not expect Multi-Column Sketches to be of use for multiple numerical attributes which are queried frequently together.

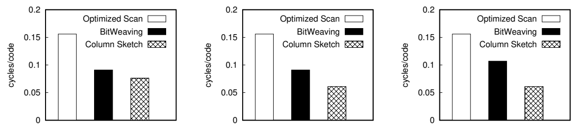
Building Multi-Column Sketches. Multi-Column Sketches are constructed via sampling much like a regular Column Sketch, with the sample being sorted if order is necessary. For the purposes of frequent values and unique codes, values are considered frequent only if the joint tuple of values from each column is above the threshold $\frac{1}{b}$. For example, a Multi-Column Sketch on (Job, City) would consider the joint value ("Software Engineer", "San Francisco") for frequent item status but not "Software Engineer" or "San Francisco" individually.

Modeling Multi-Column Sketches. The derivation of the model for Multi-Column Sketches is similar to the analysis for single column Column Sketches. Let $b_1, b_2, \ldots, b_m$ be the base columns with base value element sizes of $B_{b_1}, \ldots, B_{b_m}$. If the columns are stored in columnar fashion, with each column in a separate memory region, then the bytes touched per value is:

$$C_{col} = B_s + \sum_{i=1}^{m} B_{b_i}[1 - (1 - \frac{1}{2^{B_s}})^{\frac{M_g}{B_{b_i}}}]$$

For data which is stored in a column group, let $B_G = \sum_{i=1}^{m} B_{b_i}$ be the data size of a column group value. This value is strictly larger than the size of each individual attribute's data value, and so any given memory region of size M_g contains fewer tuples. In particular, for a region size M_g, we have $\lceil \frac{M_g}{B_G} \rceil$ tuples per region. It follows that the bytes touched per value is

$$C_{cg} = B_s + B_G[1 - (1 - \frac{1}{2^{B_s}})^{\frac{M_g}{B_G}}]$$



(a) Uniform Columnar **(b)** Uniform Column Group **(c)** Zipfian Column Group

Figure C.1: Column Sketches retain an edge over the state-of-the-art across different data layouts.

Since $\frac{M_g}{B_G} < \frac{M_g}{B_{b_i}}$ for all i, then $C_{cg} < C_{col}$ and so a Multi-Column sketch over a column group strictly dominates the performance of a Multi-Column group over columns which are stored separately.

Experimental Setup. In all experiments for Multi-Column Column Sketches the column group being sketched consists of three columns of 8 bit dictionary encoded values. The values in each column are generated independently of one another, and each column is generated using one of two distributions. In the first, the values in each column are generated using the uniform distribution. In the second, the values in each column are generated using the zipfian distribution with $z = 1.0$.

We compare the performance on equality predicates of the Multi-Column Sketch to BitWeaving and an optimized scan. For the optimized scan and the Multi-Column Sketch, the base data in the three columns is stored in either columnar format or group of columns format. For BitWeaving, the data is always stored bit-sliced.

Experimental Results. Figure C.1a shows the performance of the Column Sketch, BitWeaving, and the optimized scan over uniformly distributed data and over columnar base data. The optimized scan is bandwidth bound and gets a performance of 0.156 cycles per code. The BitWeaving scan is similar to a BitWeaving scan over a single column with 24 bits, and so the performance is 0.091 cycles/code, matching Section 5.7. The Multi-Column Sketch scan sees some overhead compared to the equality predicates conducted in Section 5.7.3. This is expected; the small value sizes mean there are more values per cacheline, and so a greater chance of some code in any cacheline matching the query endpoint. Then, for each matching code, we need to go check each value in the base data. Still, the Multi-Column Sketch is faster than BitWeaving by 20% and faster than the optimized scan by 2×.

When the base data is changed to being in column group format, the number of values per cacheline of base data decreases and so more data is skipped. The result is that the performance of the Multi-Column Sketch rises by 25%, as seen in Figure C.1b, bringing its speedup over BitWeaving to 50% and over the optimized scan to 2.8×. Finally, Figure C.1c shows the same robustness to data distribution holds over Multi-Column Sketches, with the Multi-Column Sketch having the same performance over the zipfian dataset.

C.2 Persistent Storage Systems

We can use the modeling of Section 5.5 to model how a Column Sketch would perform over systems aimed at persistent storage. Compared to in-memory systems, the main change is the large increase in the granularity of data access from $M_g = 64$ to $M_g = 4K$.

We start with the derived model for performance for a single endpoint query from Section 5.5: $Cost = B_s + B_b[1 - (1 - \frac{1}{2^{8B_s}})^{\frac{M_g}{B_b}}]$ Previously, for $M_g = 64$ we had that with $B_s = 1, B_b = 4$, the amount of bytes touched was 1.24. With those parameters, the value of $1 - (1 - \frac{1}{2^{8B_s}})^{\frac{M_g}{B_b}}$ is 0.12. By moving to disk based page sizes, this value is 0.98, and thus we have almost no chance of skipping data pages. However, if we move to two byte column sketches, our chance of skipping base data accesses becomes quite high. With $B_s = 2, B_b = 4$, we have the probability of accessing any data page becomes only 0.02, and our expected number of bytes touched per value is 2.08. For columns with element size $B_b = 8$, this chance of accessing any disk page is 0.992 and our number of bytes touches per value is 2.06.

Because the Column Sketch needs to be two bytes to be of use, we expect Column Sketches aimed at numerical attributes to improve select operator performance for stable storage based systems, regardless of whether the base column is in stable storage or in memory. In contrast, we do not expect Column Sketches aimed at a single categorical attribute to be of use to disk-based systems unless both the Column Sketch and base data column are in memory. This is because categorical attributes tend to be below or around 16 bits in length.

C.3 Performance Model: Accounting for Unique Codes

In Section 5.5, we explained that uniquely encoding all values which appear with frequency greater than $\frac{1}{2^{B_b}}$ improved the expected select performance over both uniquely encoded and non-uniquely encoded endpoints. Here, we go into more detail about how uniquely encoding values affects Column Sketch performance.

The performance model for unique codes is trivial and says that we simply touch B_s bytes. For non-unique codes, the logic is similar to Section 5.5. The main difference is that we need to update the chance that a value takes on any given non-unique endpoint. Under the assumption that there are no unique codes, the chance any value takes on the given non-unique endpoint code is $1/2^{8B_s}$. To account for unique codes, we introduce the variables n and f_n, with n the number of non-unique codes, and f_n the portion of values which are encoded by non-unique codes. For n, f_n we have $0 \leq f_n \leq 1$ and $1 \leq n \leq 2^{8B_s}$.

If we assume that each non-unique code contains roughly the amount of values as its expectation, then the chance that any given value takes on a given non-unique endpoint is simply $\frac{f_n}{n}$. The chance we skip any given cacheline is then $1 - (1 - \frac{f_n}{n})^{\frac{M_g}{B_b}}$, and that the number of bytes touched per value is

$$B_s + B_b[1 - (1 - \frac{f_n}{n})^{\frac{M_g}{B_b}}]$$

C.4 Frequently Queried Values

To model the cost of frequently queried values, we continue from the work presented in Appendix C.3. The eventual optimization problem is provably NP-hard, and so we provide solutions to this problem based on heuristics.

Let U be thet set of unique codes, and C be the total number of codes we want to assign. For each code $c \in C$, we have the cost of querying c as

$$\begin{cases} B_s & \text{if} \quad c \in U \\ B_s + B_b[1 - (1 - f_c)^{\frac{M_g}{B_b}}] & \text{if} \quad c \notin U \end{cases}$$

To optimize the cost of a given workload over a dataset, we start by incorporating into our model the variable q_c, with q_c representing the portion of queries that query on code c. Our total workload cost is then:

$$\sum_{c \in U} q_c B_s + \sum_{c \notin U} q_c \left(B_s + B_b[1 - (1 - f_c)^{\frac{M_g}{B_b}}] \right)$$

Assuming our goal is to minimize this value over all assignments of our codes, then we can simplify this equation to:

$$\sum_{c \notin U} q_c \left(1 - (1 - f_c)^{\frac{M_g}{B_b}} \right)$$

We now provide initial steps toward optimizing this assignment, with further optimization part of future work. Before delving into detail, we note the following: first, an analytical solution to this process depends on a product of terms including estimates of q_v and f_v, the proportion of all queries and data values that are on some value v. To properly estimate these terms to sufficient accuracy such that their product is accurate, significantly more samples will be needed than in Section 5.3. Second, if we let $\frac{M_g}{B_b} = 1$, then we have as a subproblem the optimization form of the partitioning problem, which is known to be NP-hard.

Unordered Column Sketches. In unordered Column Sketches, we have no dependencies between values and their given codes. Thus, we start by taking the codes which cost the most according to our cost model and making them unique. To do so, sort the values on $q_v(1 - (1 - f_v)^{\frac{M_g}{B_b}})$, which we will denote as expression $g(v)$. An optimal solution to the problem will assign unique codes to some m values which have the largest result on g. Let $q_N = 1 - \sum_{c \in U} q_c$ and $f_N = 1 - \sum_{c \in U} f_c$. Then, to estimate the number m which is optimal, we assign values to the set of unique codes U as long as the expression

$$q_N(1 - (1 - \frac{f_N}{C - |U|})^{\frac{M_g}{B_b}})$$

decreases. After, we continue along the list of values sorted on $g(v)$ and greedily assign the remaining values to the non-unique code which has the lowest value of

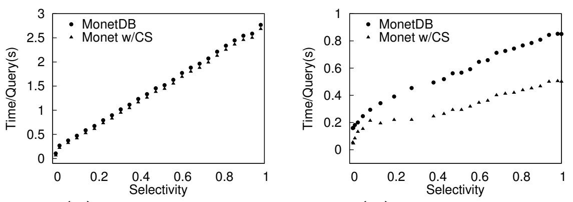
$$q_c \left(1 - (1 - f_c)^{\frac{M_g}{B_b}} \right)$$

Ordered Column Sketches. In the case that we have a number of consecutive codes that are non-unique, the optimal solution tries to place partition points into the sorted list of values such that

$$g(c) = q_c \left(1 - (1 - f_c)^{\frac{M_g}{B_b}} \right)$$

is relatively equal for all codes. Given a sorted list of n values with q_i, f_i values for each, this can be done in order of the values by deciding which partitions shift their endpoints over by a code. For instance, if have 256 partitions, upon reading a value v, the algorithm could decide to shift partitions 130-256 over by a single value, and leave the endpoints of partitions 101-129 unchanged. The decision of which code gains a new value is made greedily; for every new value which we examine, the code which sees the smallest increase in g by accepting a value does so.

We start the procedure for order-preserving Column Sketches by assuming all C codes are non-unique and performing this procedure, taking time $O(nC)$. We keep track of the value in each code C which has the largest result for $g(v)$. We then proceed to examine the value in each code c with largest $g(v)$. If giving v the unique code c and shifting the other values into codes $c - 1$ and



(a) TPC-H Q1 **(b)** TPC-H Q6

Figure C.2: Column Sketches improve scan performance on TPC-H queries

$c+1$ produces a decrease in

$$\sum_{c \notin U} q_c \left(1 - (1 - f_c)^{\frac{M_g}{B_b}}\right)$$

then we give v unique code c. Otherwise, we do not. Additionally, this procedure is again barred from giving consecutive codes unique values, and cannot give the first or last codes unique values.

C.5 Shifting Domains

In order to retain robust performance, the compression map needs to retain a relatively even number of values assigned to each code. The simplest way to do this is to count the number of values assigned to each code, and re-encode when some code is both non-unique and has too large a proportion of the dataset. Counting the number of values assigned to each code is easily done, as it can be done in one pass as data is ingested or as the Column Sketch is created. Additionally, this doesn't take too much space as tracking the number of values assigned to each code requires a single integer per code. Since the number of codes is usually quite small, this is a small memory overhead.

When re-encoding the Column Sketch, the process is similar to its original creation and involves a sampling phase followed by an encoding of the base column. Because the Column Sketch is a secondary index, this process can be done in the background and does not halt query processing. Additionally, the database can choose to use the pre-existing Column Sketch in the interim, or it can drop it immediately.

Finally, we look at the case of clustered data such as date columns and similar. First, we note that columns with clustered data usually do well with lightweight indices such as Zone Maps or Column Imprints, and so there exist prior techniques which better suit these scenarios. However, Column Sketches should retain there robust behavior in these scenarios and so we provide two ways to deal with correlated data. The first is to perform horizontal partitioning per some amount of data. The second technique is to run a regression on column order and column position at the initial creation time of a Column Sketch. If the correlation is high, then we leave some number of codes at the end of a Column Sketch empty. The number of codes to leave empty, e, is a tunable parameter. The Column Sketch scan will perform on average like each non-unique code has $\frac{1}{256-e}$ of the data, and the Column Sketch will need to re-encode the Column Sketch every time the base data reaches $\frac{256}{256-e}\times$ its prior size.

C.6 TPC-H

To run TPC-H queries we integrated Column Sketches into MonetDB. The results support the main observations from the synthetic workload experiments, with Column Sketches providing a large boost in scan performance.

To perform the integration in MonetDB, we introduced a new select operator for Column Sketches that takes as input the same API as the standard MonetDB select, i.e., a single column and a predicate. However, instead of the usual output of MonetDB which is a position list, Column Sketches outputs a bitvector. Thus, we also wrote a new fetch operator to use instead of the original fetch operator in MonetDB that works with position lists. The rest of the operators used in the TPC-H plans are the original MonetDB operators.

Furthermore, we did not do any changes in the MonetDB optimizer to use those new operators automatically. Instead, we took the plans created from the explain command in MonetDB and edited those plans to use Column Sketches. Then we fed those plans directly into the MAL interface of MonetDB. This means our results do not include the optimizer cost, and so for fairness we also remove this cost from MonetDB.

Overall, we setup these experiments by varying selectivity and we compare plain MonetDB against MonetDB with ColumnSketches (Monet w/CS) enabled. We use TPC-H scale factor 100 and provide performance experiments against query one (Q1) and query 6 (Q6) of TPC-H. The results can be seen in Figure C.2.

For both queries, Column Sketches improves on the scan performance of MonetDB. In Q1, this improvement is less apparent as the majority of the time spent in query execution is spent doing aggregation. In contrast, Q6 sees a much larger performance improvement as much more time is spent doing predicate evaluation. For both queries, this improvement is roughly constant across all selectivites. As a percentage of query time, the improvement is largest for low selectivities and smaller for higher selectivities. The improvement for Q1 ranges from 19% at 1% selectivity to 3% at 98% selectivity. The improvement for Q6 ranges from 54% at 1% selectivity to 41% at 98% selectivity.

Appendix D

Appendix: Data Calculator Primitives

			Unless otherwise specified, we use a reduced default values domain of 100 values for integers, 10 values for doubles, and 1 value for functions.		Hash Table			B+Tree/CSB+Tree/FAST			
						LPL					
		Primitive	**Domain**	**size**	**H**	**LL**	**UDP**	**B+**	**CSB+**	**FAST**	**ODP**
Node organization	1	**Key retention.** *No:* node contains no real key data, e.g., intermediate nodes of b+trees and linked lists. *Yes:* contains complete key data, e.g., nodes of b-trees, and arrays. *Function:* contains only a subset of the key, i.e., as in tries.	yes | no | function(func)	3	no	no	yes	no	no	no	yes
	2	**Value retention.** *No:* node contains no real value data, e.g., intermediate nodes of b+trees, and linked lists. *Yes:* contains complete value data, e.g., nodes of b-trees, and arrays. *Function:* contains only a subset of the values.	yes | no | function(func)	3	no	no	yes	no	no	no	yes
	3	**Key order.** Determines the order of keys in a node or the order of fences if real keys are not retained.	none | sorted | k-ary (k: int)	12	none	none	none	sorted	sorted	4-ary	sorted
	4	**Key-value layout.** Determines the physical layout of key-value pairs. Rules: requires key retention != no or value retention != no.	row-wise | columnar | col-row-groups(size: int)	12			col.				col.
	5	**Intra-node access.** Determines how sub-blocks (one or more keys of this node) can be addressed and retrieved within a node, e.g., with direct links, a link only to the first or last block, etc.	direct | head_link | tail_link | link_function(func)	4	direct	head	direct	direct	direct	direct	direct
	6	**Utilization.** Utilization constraints in regards to capacity. For example, >= 50% denotes that utilization has to be greater than or equal to half the capacity.	= (X%) | function(func) | none *(we currently only consider X=50)*	3	none	none	none	>= 50%	>= 50%	>= 50%	none
Node filters	7	**Bloom filters.** A node's sub-block can be filtered using bloom filters. Bloom filters get as parameters the number of hash functions and number of bits.	off | on(num_hashes: int, num_bits: int) *(up to 10 num_hashes considered)*	1001	off	off	off	off	off	off	off
	8	**Zone map filters.** A node's sub-block can be filitered using zone maps, e.g., they can filter based on mix/max keys in each sub-block.	min | max | both | exact | off	5	off	off	off	min	min	min	off
	9	**Filters memory layout.** Filters are stored contiguously in a single area of the node or scattered across the sub-blocks. Rules: requires bloom filter != off or zone map filters != off.	consolidate | scatter	2				scatter	scatter	scatter	
Partitioning	10	**Fanout/Radix.** Fanout of current node in terms of sub-blocks. This can either be unlimited (i.e., no restriction on the number of sub-blocks), fixed to a number, decided by a function or the node is terminal and thus has a fixed capacity.	fixed(value: int) | function(func) mited | terminal(cap: int) *(up to 10 different capacities and up to 10 fixed fanout values are considered)*	22	fixed(100)	unlimited	term(256)	fixed(20)	fixed(20)	fixed(16)	term(256)
	11	**Key partitioning.** Set if there is a pre-defined key partitioning imposed. e.g. the sub-block where a key is located can be dictated by a radix or range partitioning function. Alternatively, keys can be temporaly partitioned. If partitioning is set to none, then keys can be forward or backwards appended.	none(fw-append | bw-append) | range() | radix() | function (func) | temporal(size_ratio: int, merge_policy: [tier| level])	205	range(100)	none(fw)	none(fw)	none(fw)	none(fw)	none(fw)	none(fw)
	12	**Sub-block capacity.** Capacity of each sub-block. It can either be fixed to a value, or balanced (i.e., all sub-blocks have the same size), unrestricted or functional. Rules: requires key partitioning != none.	fixed(value: int) | balanced | stricted | function(func) *(up to 10 different fixed capacity values are considered)*	13	unrestrict.	fixed(256)		balanced	balanced	balanced	
	13	**Immediate node links.** Whether and how sub-blocks are connected.	next | previous | both | none	4	none	next	none	none	none	none	none
	14	**Skip node links.** Each sub-block can be connected to another sub-block (not only the next or previous) with skip-links. They can be perfect, randomized or custom.	perfect | randomized(prob: double) | function(func) | none	13	none	none	none	none	none	none	none
	15	**Area-links.** Each sub-tree can be connected with another sub-tree at the leaf level throu area links. Examples include the linked leaves of a B+Tree.	forward | backward | both | none	4	none	none	forw.	none	none	none	none
Children layout	16	**Sub-block physical location.** This represents the physical location of the sub-blocks. Pointed: in heap, Inline: block physically contained in parent. Double-pointed: in heap but with pointers back to the parent. Rules: requires fanout/radix != terminal.	inline | pointed | double-pointed	3	pointed	inline		pointed	pointed	pointed	
	17	**Sub-block physical layout.** This represents the physical layout of sub-blocks. Scatter: random placement in memory. BFS: laid out in a breadth-first layout. BFS layer list: hierarchical level nesting of BFS layouts. Rules: requires fanout/radix != terminal.	BFS | BFS layer(level-grouping: int) | scatter *(up to 3 different values for layer-grouping are considered)*	5	scatter	scatter		scatter	BFS	BFS-LL	
	18	**Sub-blocks homogeneous.** Set to true if all sub-blocks are of the same type. Rules: requires fanout/radix != terminal.	boolean	2	true	true		true	true	true	
	19	**Sub-block consolidation.** Single children are merged with their parents. Rules: requires fanout/radix != terminal.	boolean	2	false	false		false	false	false	
	20	**Sub-block instantiation.** If it is set to eager, all sub-blocks are initialized, otherwise they are initialized only when data are available (lazy). Rules: requires fanout/radix != terminal.	lazy | eager	2	lazy	lazy		lazy	lazy	lazy	
	21	**Sub-block links layout.** If there exist links, are they all stored in a single array (consolidate) or spread at a per partition level (scatter). Rules: requires immediate node links != none or skip links != none.	consolidate | scatter	2		scatter					
Recursion	22	**Recursion allowed.** If set to yes, sub-blocks will be subsequently inserted into a node of the same type until a maximum depth (expressed as a function) is reached. Then the terminal node type of this data structure will be used. Rules: requires fanout/radix != terminal.	yes(func) | no	3	no	no		yes(logn)	yes(logn)	yes(logn)	

Total number of property combinations > 10^18 / 60 invalid combinations ≈ 10^16

Node descriptions: H : Hash, **LL**: Linked List, **LPL**: Linked Page-List, **UDP**: Unordered Data Page, **B+**: B+Tree Internal Node
CSB+: CSB+Tree Internal Node, **FAST**: FAST Internal node, **ODP**: Ordered Data Page (Nodes highlighted with gray are terminal leaf nodes)

Figure D.1: Data layout primitives and synthesis examples of data structures.

Data Access Primitives and Fitted Models

	Data Access Primitives Level 1 (required parameters ; optional parameters)	**Model Parameters**	**Data Access Primitives Layer 2**	**Fitted Models**
1	**Scan** (Element Size, Comparison, Data Layout; None)	Data Size	Scalar Scan (RowStore, Equal)	Linear Model (1)
2			Scalar Scan (RowStore, Range)	Linear Model (1)
3			Scalar Scan (ColumnStore, Equal)	Linear Model (1)
4			Scalar Scan (ColumnStore, Range)	Linear Model (1)
5			SIMD-AVX Scan (ColumnStore, Equal)	Linear Model (1)
6			SIMD-AVX Scan (ColumnStore, Range)	Linear Model (1)
7	**Sorted Search** (Element Size, Data Layout;)	Data Size	Binary Search (RowStore)	Log-Linear Model (2)
8			Binary Search (ColumnStore)	Log-Linear Model (2)
9			Interpolation Search (RowStore)	Log + LogLog Model (3)
10			Interpolation Search (ColumnStore)	Log + LogLog Model (3)
11	**Hash Probe** (; Hash Family)	Structure Size	Linear Probing (Multiply-shift Dietzfelbinger et al. [1997])	Sum of Sigmoids (5), Weighted Nearest Neighbors (7)
12			Linear Probing (k-wise independent, k=2,3,4,5)	Sum of Sigmoids (5), Weighted Nearest Neighbors (7)
13	**Bloom Filter Probe** (; Hash Family)	Structure Size, Number of Hash Functions	Bloom Filter Probe (Multiply-shift Dietzfelbinger et al. [1997])	Sum of Sum of Sigmoids (6), Weighted Nearest Neighbors (7)
14			Bloom Filter Probe (k-wise independent, k=2,3,4,5)	Sum of Sum of Sigmoids (6), Weighted Nearest Neighbors (7)
15	**Sort** (Element Size; Algorithm)	Data Size	QuickSort	NLogN Model (4)
16			MergeSort	NLogN Model (4)
17			ExternalMergeSort	NLogN Model (4)
18	**Random Memory Access**	Region Size	Random Memory Access	Sum of Sigmoids (5), Weighted Nearest Neighbors (7)
19	**Batched Random Memory Access**	Region Size	Batched Random Memory Access	Sum of Sigmoids (5), Weighted Nearest Neighbors (7)
20	**Unordered Batch Write** (Layout;)	Write Data Size	Contiguous Write (RowStore)	Linear Model (1)
21			Contiguous Write (ColumnStore)	Linear Model (1)
22	**Ordered Batch Write** (Layout;)	Write Data Size, Data Size	Batch Ordered Write (RowStore)	Linear Model (1)
23			Batch Ordered Write (ColumnStore)	Linear Model (1)
24	**Scattered Batch Write**	Number of Elements, Region Size	ScatteredBatchWrite	Sum of Sum of Sigmoids (6), Weighted Nearest Neighbors (7)

Models used for fitting data access primitives

	Model	**Description**	**Formula**
1	Linear	Fits a simple line through data	$f(\mathbf{v}) = \mathbf{w}^\top \phi(\mathbf{v}) + y_0; \phi(v) = (v)$
2	Log-Linear	Fits a linear model with a basis composed of the identity and logarithmic functions plus a bias	$f(\mathbf{v}) = \mathbf{w}^\top \phi(\mathbf{v}) + y_0; \phi(v) = \begin{pmatrix} v \\ \log v \end{pmatrix}$
3	Log + LogLog	Fits a model with log, $\log\log$, and linear components	$f(\mathbf{v}) = \mathbf{w}^\top \phi(\mathbf{v}) + y_0; \phi(v) = \begin{pmatrix} v \\ \log v \\ \log\log v \end{pmatrix}$
4	NLogN	Fits a model with primarily an NLogN and linear component	$f(\mathbf{v}) = \mathbf{w}^\top \phi(\mathbf{v}) + y_0; \phi(v) = \begin{pmatrix} v \log v \\ v \end{pmatrix}$
5	Sum of Sigmoids	Fits a model that has k approximate steps. Seen as sigmoids in $\log x$ as this fits various cache behaviors nicely	$f(x) = \sum_{i=1}^{k} \frac{c_i}{1+e^{-k_i(\log x - x_i)}} + y_0$
6	Sum of Sum of Sigmoids	Fits a model which has two cost components, both of which have k approximate steps occuring at the same locations.	$f(x, m) = \sum_{i=1}^{k} \frac{c_{i1}}{1+e^{-k_i(\log x - x_i)}} + (m-1)(\sum_{i=1}^{k} \frac{c_{i2}}{1+e^{-k_i(\log x - x_i)}} + y_1) + y_0$
7	Weighted Nearest Neighbors	Takes the k nearest neighbors under the l_2 norm and computes a weighted average of their outputs. The input x is allowed to be a vector of any size.	Let $x_1, \ldots x_k$ be the nearest neighbors of x with costs $y_1, \ldots, y_k$. Then $f(x) = \frac{1}{\sum_{i=1}^{k} \frac{1}{d(x, x_i)}} \sum_{i=1}^{k} \frac{1}{d(x, x_k)} y_k$

Notation: f is a function, $\mathbf{v}$ is a vector, and x, m are scalars. $\log(\mathbf{v})$ returns a vector with log applied on an element by element basis to $\mathbf{v}$; i.e. if $v = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$, then $\log v = \begin{pmatrix} \log v_1 \\ \log v_2 \end{pmatrix}$. Finally, if we have vectors $v^{(1)}, v^{(2)}$ of lengths n, m stacked on each other as $\begin{pmatrix} v^{(1)} \\ v^{(2)} \end{pmatrix}$, then this signifies the $n+m$ length vector produced by stacking the entries of $v^{(1)}$ on top of the entries of $v^{(2)}$; i.e. $\begin{pmatrix} v^{(1)} \\ v^{(2)} \end{pmatrix} = \left(v_1^{(1)}, \ldots, v_n^{(1)}, v_1^{(2)}, \ldots, v_m^{(2)}\right)^\top$.

Table D.1: Data access primitives and models used for operation cost synthesis.